Continuous Thought Machines

Luke Darlow[1], **Ciaran Regan**[1,2], **Sebastian Risi**[1,3], **Jeffrey Seely**[1] **and Llion Jones**[1]
[1]Sakana AI, [2]University of Tsukuba, [3]IT University of Copenhagen

Biological brains demonstrate complex neural activity, where the timing and interplay between neurons is critical to how brains process information. Most deep learning architectures simplify neural activity by abstracting away temporal dynamics. In this paper we challenge that paradigm. By incorporating neuron-level processing and synchronization, we can effectively reintroduce neural timing as a foundational element. We present the Continuous Thought Machine (CTM), a model designed to leverage neural dynamics as its core representation. The CTM has two core innovations: (1) **neuron-level temporal processing**, where each neuron uses unique weight parameters to process a history of incoming signals; and (2) **neural synchronization employed as a latent representation**. The CTM aims to strike a balance between oversimplified neuron abstractions that improve computational efficiency, and biological realism. It operates at a level of abstraction that effectively **captures essential temporal dynamics while remaining computationally tractable** for deep learning. We demonstrate the CTM's strong performance and versatility across a range of challenging tasks, including ImageNet-1K classification, solving 2D mazes, sorting, parity computation, question-answering, and RL tasks. Beyond displaying rich internal representations and offering a natural avenue for interpretation owing to its internal process, the CTM is able to perform tasks that require complex sequential reasoning. The CTM can also leverage adaptive compute, where it can stop earlier for simpler tasks, or keep computing when faced with more challenging instances. The goal of this work is to share the CTM and its associated innovations, rather than pushing for new state-of-the-art results. To that end, we believe the CTM represents a significant step toward developing more biologically plausible and powerful artificial intelligence systems.

Alongside this report, we release the CTM code repository that includes the model checkpoints. We also encourage the reader to view our project page for interactive demonstrations to best highlight the CTM's capabilities.

Corresponding author(s): Luke Darlow (luke@sakana.ai)

Contents

1. Introduction

Neural networks (NNs) were originally inspired by biological brains, yet they remain significantly distinct from their biological counterparts. Brains demonstrate complex neural dynamics that evolve over time, but modern NNs intentionally abstract away such temporal dynamics in order to facilitate large-scale deep learning. For instance, the activation functions of standard NNs can be seen as an intentional abstraction of a neuron's firing rate, replacing the temporal dynamics of biological processes with a single, static value. Such simplifications, though enabling significant advancements in large-scale machine learning (Goodfellow et al., 2016; LeCun et al., 2015; Wei et al., 2022), have resulted in a departure from the fundamental principles that govern biological neural computation.

Over hundreds of millions of years, evolution has endowed biological brains with rich neural dynamics, including spike-timing-dependent plasticity (STDP) (Caporale and Dan, 2008) and neuronal oscillations. Emulating these mechanisms, particularly the temporal coding inherent in spike timing and synchrony, presents a significant challenge. Consequently, modern neural networks do not rely on temporal dynamics to perform compute, but rather prioritize simplicity and computational efficiency. This abstraction, while boosting performance on specific tasks, contributes to a recognized gap between the flexible, general nature of human cognition and current AI capabilities, suggesting fundamental components, potentially related to temporal processing, are missing from our current models (Chollet, 2019; Lake et al., 2017; Marcus, 2018).

Why do this research? Indeed, the notably high performance of modern AI across many practical fields suggests the emulation of neural dynamics is unwarranted, or that explicitly accounting for temporal aspects of intelligence is anti-pragmatic. However, human intelligence is highly flexible, data-efficient, is fluid in that it extrapolates well to unseen circumstances, and exists in an open world where learning and adaptation are tied to the arrow of time. Consequently, human intelligence includes common sense, the ability to leverage ontological reasoning, transparency/explainability, and strong generalization. AI does not yet convincingly display these properties (Chollet, 2019; Hohenecker and Lukasiewicz, 2020; Marcus, 2018; Thompson et al., 2020).

For these reasons, we argue that time should be a central component of artificial intelligence in order for it to eventually achieve levels of competency that rival or surpass human brains (Cariani and Baker, 2022; Maass, 2001). Therefore, in this work, we address the strong limitation imposed by overlooking neural activity as a central aspect of intelligence. We introduce the *Continuous Thought Machine* (CTM), a novel neural network architecture designed to explicitly incorporate neural timing as a foundational element. Our contributions are as follows:

1. We introduce an **decoupled internal dimension**, a novel approach to modeling the temporal evolution of neural activity. We view this dimension as that over which thought can unfold in an artificial neural system, hence the choice of nomenclature. While internal recurrence is not a new concept in neural networks, our innovation lies in leveraging this recurrence to explicitly construct and manipulate neural activity patterns. By progressing through discrete internal ticks (Kirsch and Schmidhuber, 2021; Kirsch et al., 2022; Pedersen et al., 2024; Schwarzschild et al., 2021), this internal dimension allows the CTM to build up complex, time-dependent neural dynamics, directly addressing the biological principle that the timing of neural events is crucial for neural computation. This contrasts with traditional uses of recurrence, which primarily focus on processing sequential data rather than generating dynamic neural activity.
2. We provide a mid-level abstraction for neurons, which we call **neuron-level models** (NLMs), where every neuron has its own internal weights that process a history of incoming signals (i.e., pre-activations) to calculate its next activation (as opposed to a static ReLU, for example). This approach is simple to implement, scales well with existing deep learning architectures, and

results in the emergence of complex neural activation dynamics (i.e., neural activity), exhibiting a higher degree of variability than static activation functions (see Sections 3.2 and 5).
3. We use **neural synchronization** directly as the latent representation with which the CTM observes (e.g., through an attention query) and predicts (e.g., via a projection to logits). This biologically-inspired design choice puts forward neural activity as the crucial element for any manifestation of intelligence the CTM might demonstrate.

Reasoning models and recurrence. The frontier of artificial intelligence faces a critical juncture: moving beyond simple input-output mappings towards genuine reasoning capabilities. While scaling existing models has yielded remarkable advancements, the associated computational cost and data demands are unsustainable and raise questions about the long-term viability of this approach. For sequential data, longstanding recurrent architectures (Dey and Salem, 2017; Hochreiter and Schmidhuber, 1997; Medsker and Jain, 1999) have largely been superseded by transformer-based approaches (Vaswani et al., 2017). Nevertheless, recurrence is re-emerging as a natural avenue for extending model complexity. Recurrence is promising because it enables iterative processing and the accumulation of information over time. Modern text generation models use intermediate generations as a form of recurrence that enables additional compute during test-time. Recently, other works have demonstrated the benefits of the recurrent application of latent layers (Geiping et al., 2025; Jaegle et al., 2021; Yang et al., 2023).

While such methods bring us closer to the recurrent structure of biological brains, a fundamental gap nevertheless remains. **We posit that recurrence, while essential, is merely one piece of the puzzle.** The temporal dynamics unlocked by recurrence – the precise timing and interplay of neural activity – are equally crucial. The CTM differs from existing approaches in three ways: (1) the internal 'thought' dimension enables sequential thought on any conceivable data modality; (2) private neuron-level models enables the consideration of precise neural timing; and (3) neural synchronization used directly as a representation for solving tasks.

Useful side effects The CTM's internal recurrence is analogous to thought (hence the chosen nomenclature), where it can stop 'thinking' earlier for simpler tasks (e.g., an easily identifiable image; see Figure 5), or go deeper for more challenging tasks (e.g., a long maze; see Section 4.3), thus enabling a form of adaptive compute. In particular, the CTM achieves a kind of adaptive computation without the need for additional losses that are difficult to tune (Graves, 2016). Indeed, we observe the emergence of interpretable and intuitive problem-solving strategies, suggesting that leveraging neural timing can lead to more emergent benefits and potentially more effective AI systems. Another positive effect of being explicit about neural timing is that information can be encoded within this timing, the result of which is a greater capacity for contextualization – we design our 2D maze solving challenge to test this (Section 4).

The rest of this paper is structured as follows. In Section 2 we give the technical details of the CTM. In Section 3 through 10 we apply the CTM to image classification, 2D mazes, sort, parity, question-answering, and simple reinforcement learning tasks. Each experiment is designed to investigate specific characteristics and we compare to baselines wherever possible. In Section 12 we discuss our findings, suggesting avenues for future work. We draw final conclusions in Section 13. By explicitly modeling neural timing through the CTM, we aim to pave the way for more biologically plausible and performant artificial intelligence systems.

2. Method

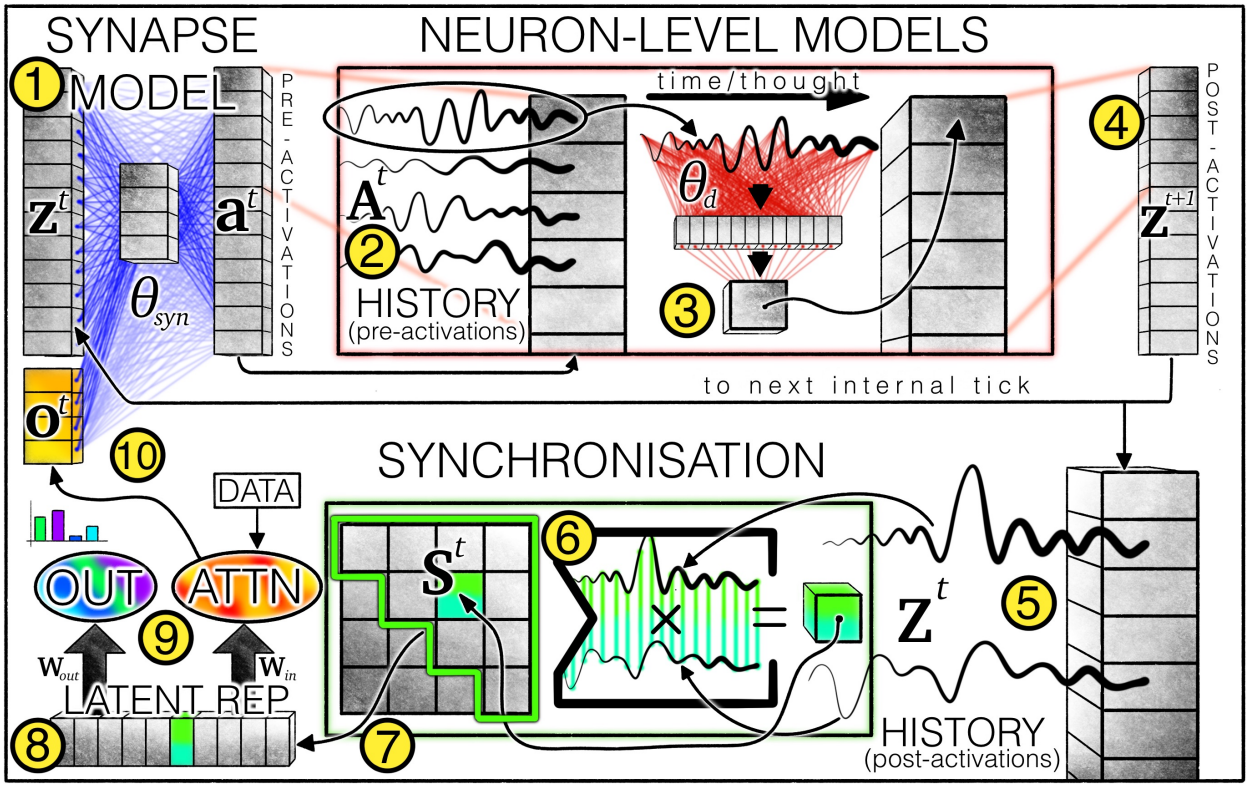


Figure 1 | CTM architecture overview. The ① synapse model (weights depicted as blue lines; Equation 1) models the cross-neuron interactions to produce pre-activations. For each neuron, a ② history of pre-activations is kept (Equation 2), the most recent of which are used by the ③ neuron-level models (weights depicted as red lines, Equation 3) to produce ④ post-activations. A ⑤ history of post-activations is also kept (Equation 4) and used to ⑥ compute a synchronization matrix (Equation 5 and Equation 10). Neuron pairs are ⑦ selected (see Section 2.4.1) from the synchronization matrix, yielding the ⑧ latent representations with which the CTM ⑨ produces outputs (Equation 6) and modulates data through cross-attention (Equation 7 and 8). Modulated data (e.g., attention outputs) are ⑩ concatenated with post-activations for the next internal tick.

The Continuous Thought Machine (CTM) is a neural network architecture that enables a novel approach to thinking about data. It departs from conventional feed-forward models by explicitly incorporating the concept of **neural dynamics** as the central component to its functionality. Figure 1 gives an overview of the CTM, with numbers ① to ⑩ designating flow, and Listing 1 gives a simplified overview listing for clarity. The yellow numbers in Figure 1 will be referred to throughout the rest of this paper.

Other recurrent architectures (e.g., RNNs) can be set up to incorporate an internal time dimension that is separate from data (Graves, 2016; Kirsch and Schmidhuber, 2021; Kirsch et al., 2022; Pedersen et al., 2024; Schwarzschild et al., 2021), but the CTM differs in two crucial ways: (1) instead of using conventional activation functions, the **CTM applies neuron-level models**, each with their own weights, to histories of pre-activations in order to produce complex neuron-level activity (see Section 3 for examples); and (2) the CTM uses **neural synchronization directly** as the latent representation when modulating data and producing outputs (see Section 2.4), effectively enabling a new depth of capacity where it creates, maintains, and leverages the exact timing and interplay of neurons. In the following subsections we will describe the CTM in detail.

```
################## DEFINITIONS  (hyper parameters not shown for simplicity) ################
# Backbone can be, e.g., a ResNet for images
backbone = FeatureEncoder()
# Q and KV projectors, and standard attention module
q_projector, kv_projector = Linear(), Linear()
attn = MultiHeadAttention()
# Synapse model can be linear or MLP or U-NET
synapses = MLP()
# Neuron level models (see Listing 2)
neuron_level_models = NLMS()
# Output projector from synchronisation (see Listing 3)
output_proj = Linear()
# Initialise pre-activations and z as learnable parameters
# D is the model width
z_init = Parameter(size=(D))
# M is the neuron memory length
pre_acts_history_init = Parameter(size=(D, M))

######################################## MODEL LOGIC ########################################
# Featurise inputs with backbone and compute KV tokens
kv = kv_projector(backbone(inputs))
# In each minibatch, initialise the learnable pre_act_history
pre_acts_history = pre_acts_history_init.unsqueeze(0).repeat(B, dim=0)  # (B, D, M)
# And start the post_acts_history with the learnable z_init
post_acts_history = [z_init.unsqueeze(0).repeat(B, dim=0)]
outputs_history = []
# Get initial action synchronisation to query data
synch_a = compute_synch(post_acts_history, type="action")
# Other initialisations, including learnable start histories and first pre-attn round
for step in range(n_thought_steps):
    # Project attention query off synchronisation
    q = q_projector(synch_a)
    attn_out = attn(q, kv, kv)
    # Concatenate attention output and process via synapses
    pre_acts = synapses(concat((attn_out, z)))
    # Keep history of pre-activations. This is a FIFO structure:
    pre_acts_history = concat((pre_acts_history[:, :, :-1], pre_acts), dim=-1)
    # Compute post-activations using histories (see Listing 2)
    z = neuron_level_models(pre_acts_history)
    post_acts_history.append(z)
    # Compute synchronisations (see Listing 3)
    synch_a = compute_synch(post_acts_history, type="action")
    synch_o = compute_synch(post_acts_history, type="output")
    # Projecte prediction/output off synchronisation
    outputs_history.append(output_proj(synch_o))
# Return outputs per thought step for loss function
return outputs_history
```

Listing 1 | Simplified overview of the CTM code. Features are encoded using a backbone (e.g., ResNet layers for images), data is attended to by ⑨ projecting a query from neural synchronization, information is shared across neurons using an ① MLP synapse model to produce pre-activations, ③ private neuron-level models are applied to a ② tracked history of pre-activations (see Listing 2), synchronization is computed from a ⑤ tracked history of post-activations (see Listing 3), and outputs are ⑨ projected off of synchronization. All code is available here.

2.1. Continuous thought: the internal sequence dimension

We start by introducing the internal dimension over which cognition can occur: $t \in \{1, \ldots, T\}$, a single step of which is shown as the flow from ① to ⑩. This dimension is decoupled from the data in that it unfolds internally and is not tied to any data dimensions. Recurrence along an internal dimension is by no means a new concept (Chahine et al., 2023; Geiping et al., 2025; Jaeger, 2007), and it is garnering increased focus owing to a recent drive to build *reasoning capabilities* into modern AI. Unlike conventional sequential models – such as RNNs or Transformers – that process inputs step-by-step according to the sequence inherent in the data (e.g., words in a sentence or frames in a video), the CTM operates along a self-generated timeline of internal 'thought steps.' This internal unfolding allows the model to iteratively build and refine its representations, even when processing static or non-sequential data such as images or mazes. As a result, the CTM can engage in a thought process that is decoupled from external timing, enabling more flexible, interpretable, and biologically inspired computation. To conform with existing nomenclature used in related works (Kirsch and Schmidhuber, 2021; Kirsch et al., 2022; Pedersen et al., 2024; Schwarzschild et al., 2021), we refer to these thought steps as 'internal ticks' from here on. The CTM's internal dimension is that over which the dynamics of neural activity can unfold. We believe that such dynamics are likely a cornerstone of intelligent thought. The *for loop* in Listing 1 captures the process depicted in Figure 1.

2.2. Recurrent weights: synapses

A ① synapse model, $f_{\theta_{\text{syn}}}$, interconnects the neurons of a shared D-dimensional latent space, $\mathbf{z}^t \in \mathbb{R}^D$, where t is the current internal tick in the recurrent unfolding of the CTM. θ_{syn} are the weights of the recurrent synapse model. The synapse model can be parameterized as a single linear projection followed by a standard activation function, or it can be a multi-layer perceptron (MLP). We found experimentally that a U-NET-esque (Ronneberger et al., 2015) MLP architecture performed better for all tasks (see Appendix B.1 for details of the synapse model). This indicates that synaptic connections benefit from additional, deeper compute. Applying the synapse model produces what we consider **pre-activations** at internal tick t:

$$\mathbf{a}^t = f_{\theta_{\text{syn}}}(\text{concat}(\mathbf{z}^t, \mathbf{o}^t)) \in \mathbb{R}^D, \tag{1}$$

where $\mathbf{o}^t$ is from input data (either directly or as the output of attention; see Equation 8), which we will explain in Section 2.4.

The M **most recent pre-activations** are then collected into ② a pre-activation 'history':

$$\mathbf{A}^t = \left[\mathbf{a}^{t-M+1} \quad \mathbf{a}^{t-M+2} \quad \cdots \quad \mathbf{a}^t\right] \in \mathbb{R}^{D \times M}. \tag{2}$$

The first M elements in the history and the first $\mathbf{z}^{t=1}$ need to be initialized. We initially experimented with initializing these two zeros (a similar strategy is undertaken for RNNs), but found that making these learnable parameters was best.

2.3. Privately-parameterized neuron-level models

```
# Initialisations
weights_1 = Parameter(shape=(M, d_hidden, d_model))
bias_1 = zeros(shape=(1, d_hidden, d_model))
weights_2 = Parameter(shape(d_hidden, d_model))
bias_2 = zeros(shape=(1, d_model))
# Forward pass
# b=batch, M=memory, d=d_model, h=d_hidden
# inputs are shape (b, d, M)
inputs = pre_acts_history[-M:]
out = einsum('bdM,Mhd->bdh', inputs, weights_1) + bias_1
out = einsum('bdh,hd->bd', out, weights_2) + bias_2
```

Listing 2 | Neuron-level models: ③. Using einsum greatly simplifies and speeds up the application of neuron-level models as their outputs can be computed in parallel. The first einsum computes a h-dimension latent for each neuron from the (truncated to M most recent) incoming history, ②. The second einsum then computes the single activation per-neuron (ignore the '1' dimension here for simplicity).

M effectively defines the length of the history of pre-activations that each neuron-level model works with. We tested a range of values for M, and found a range of 10-100 to be effective. Each neuron, $\{1, \ldots, D\}$, is then given its own ③ privately parameterized model that produces what we consider ④ post-activations:

$$\mathbf{z}_d^{t+1} = g_{\theta_d}(\mathbf{A}_d^t), \tag{3}$$

where θ_d are the unique parameters for neuron d, and $\mathbf{z}_d^{t+1}$ is a single unit in the vector that contains all post-activations. We use an MLP with a single hidden layer of width d_{hidden}. $\mathbf{A}_d^t$ is a M-dimensional vector (time series). Letting neurons have their own internal models does require additional parameters, scaling as $D \times (M \times H_{dim} + H_{dim})$ (where H_{dim} is the width of the neuron-level MLPs, ignoring bias parameters for simplicity). This additional parameter cost enables more modeling freedom, however.

The full set of neuron post-activations are then ⑩ concatenated with attention output (see Section 2.4) and fed recurrently into f_{θ_1} to produce pre-activations for next step, $t+1$, in the unfolding thought process. The recurrent thought-process employed by the CTM is captured in Listings 1 and 2. We will now discuss how the CTM interacts with data (both inputs and outputs) through synchronization.

2.4. Neural synchronization: modulating data and outputs

```
# AT INITIALISATION:
# Pre choose D_chosen neuron pairs from D total neurons
# D_chosen can be D_out or D_action
# Other neuron selection strategies exist, but here we show random selection
idxs_left = randint(low=0, high=D, size=D_chosen)
idxs_right = randint(low=0, high=D, size=D_chosen)  # can overlap
# Define learnable exponential decay scaling factors per neuron pair
r = Parameter(zeros(1, D_chosen, 1))
# INITIALISATION OVER.
# IN FORWARD PASS:
S = stack(post_acts_history, 1)  # S is of shape [B, T=history length]
# decay BACK in time
t_back = range(T-1, -1, -1).reshape(1, T, 1)
# Compute per NEURON PAIR exponential decays, and expand over D_chosen
exp_decay = exp(-t_back * r).expand(1, T, D_chosen)
# Compute weighted inner dot products using differet subsets of neurons
S_multiplied = S[:,:,idxs_left] * exp_decay * S[:,:,idxs_right]  # [B, T, D_chosen]
# Sum over the free T dimension and normalise by sqrt of AUC of decays
synch_representation = (S_multiplied).sum(1)/sqrt(exp_decay.sum(1))  # [B, D_chosen]
```

Listing 3 | Neural synchronization to create the latent representations used in Listing 1. Careful reshaping and broadcasting enables the use of per neuron-pair learnable exponential decays for synchronization, which lets the CTM learn complex timing dependencies. The decay parameters are initialized as zeros (i.e., no decay). This process is repeated for output and action (see Section 2.4.1). In practice we use a recursive approach that greatly reduces compute overhead; see Appendix K.

How should the CTM interact with the outside world? Specifically, how should the CTM consume inputs and produce outputs? We introduced a timing dimension over which something akin to thought can unfold. We also want the CTM's relationship with data (its interaction, so to speak) to depend not on a *snapshot* of the state of neurons (at some t), but rather on the **ongoing temporal dynamics of neuron activities**.[1] By way of solution, we turn again to natural brains for inspiration and find the concept of neural synchronization (Uhlhaas et al., 2009) both fitting and powerful. For synchronization we start by collecting the post-activations into ⑤ a post-activation 'history':

$$\mathbf{Z}^t = \begin{bmatrix} \mathbf{z}^1 & \mathbf{z}^2 & \cdots & \mathbf{z}^t \end{bmatrix} \in \mathbb{R}^{D \times t}. \tag{4}$$

The length of $\mathbf{Z}^t$ is equal to the current internal tick, meaning that **this dimension is not fixed** and can be arbitrarily large. We define neural synchronization as the matrix yielded by ⑥ the inner dot product between post-activation histories:

$$\mathbf{S}^t = \mathbf{Z}^t \cdot (\mathbf{Z}^t)^\intercal \in \mathbb{R}^{D \times D}. \tag{5}$$

2.4.1. Neuron pairing: a sub-sampling approach

Since this matrix scales in $O(D^2)$ it makes practical sense to subsample (i, j) row-column pairs, which capture the synchronization between neurons i and j. To do so we randomly select D_{out} and D_{action} (i, j) pairs from $\mathbf{S}$, thus collecting two **synchronization representations**, $\mathbf{S}^t_{\text{out}} \in \mathbb{R}^{D_{\text{out}}}$ and $\mathbf{S}^t_{\text{action}} \in \mathbb{R}^{D_{\text{action}}}$. $\mathbf{S}^t_{\text{out}}$ can then be projected to an output space as:

$$\mathbf{y}^t = \mathbf{W}_{\text{out}} \cdot \mathbf{S}^t_{\text{out}}, \tag{6}$$

and $\mathbf{S}^t_{\text{action}}$ can be used to take actions in the world (e.g., via attention as is in our setup):

$$\mathbf{q}^t = \mathbf{W}_{\text{in}} \cdot \mathbf{S}^t_{\text{action}}, \tag{7}$$

where $\mathbf{W}_{\text{out}}$ and $\mathbf{W}_{\text{in}}$ are learned weight matrices that project synchronization into vectors for observation (e.g., attention queries, $\mathbf{q}^t$) or outputs (e.g., logits, $\mathbf{y}^t$). Even though there are $(D \times (D+1))/2$ unique pairings in $\mathbf{S}^t$, D_{out} and D_{action} can be orders of magnitude smaller than this. That said, the full synchronization matrix is a large representation that has high future potential.

[1] We did begin with snapshot representations but struggled to get stable behavior owing to the emergent oscillatory behavior of neurons.

In most of our experiments we used standard cross attention (Vaswani et al., 2017):

$$\mathbf{o}^t = \text{Attention}(Q = \mathbf{q}^t, KV = \text{FeatureExtractor}(\text{data})), \tag{8}$$

where a 'FeatureExtractor' model, e.g., a ResNet (He et al., 2016), is first used to build useful local features for the keys and values. $\mathbf{o}^t \in \mathbb{R}^{d_{\text{input}}}$ is the output of attention using n_{heads} heads and is (10) concatenated with $\mathbf{z}^{t+1}$ for the next cycle of recurrence. For clarity, Listing 3 demonstrates what this process looks like in code, including learnable temporal dependency scaling (see below).

Scaling temporal dependency Since $\mathbf{S}^t$ aggregates information from all previous internal ticks up to step t, later steps could potentially have a diminishing influence on synchronization. To provide the CTM with the flexibility to modulate the influence of past activity, we introduce learnable exponentially decaying rescaling factors. Given learnable scaling factors $r_{ij} \geq 0$ for each neuron pair ij, the rescaling factors over t are computed as:

$$\mathbf{R}_{ij}^t = \left[\exp(-r_{ij}(t-1)) \quad \exp(-r_{ij}(t-2)) \quad \cdots \quad \exp(0)\right]^\intercal \in \mathbb{R}^t. \tag{9}$$

$\mathbf{R}_{ij}^t$ is then used to rescale the components of the synchronization dot product:

$$\mathbf{S}_{ij}^t = \frac{(\mathbf{Z}_i^t)^\intercal \cdot \text{diag}(\mathbf{R}_{ij}^t) \cdot \left(\mathbf{Z}_j^t\right)}{\sqrt{\sum_{\tau=1}^t \left[\mathbf{R}_{ij}^t\right]_\tau}}, \tag{10}$$

Intuitively, higher r_{ij} values result in shorter-term dependency by biasing the dot product towards more recent internal ticks, while $r_{ij} = 0$ recovers the standard dot product (we also initialise to zeros). The square-root normalization prevents any single ij combination from dominating downstream processing (Vaswani et al., 2017). In practice we only need to compute $\mathbf{R}_{ij}^t$ for subsampled synchronization pairs, which we do separately for output and action representations. Since CTM can learn to alter the decay rates across different neuron pairs, the inclusion of r into the learnable parameters effectively enhances the complexity of the CTM's reliance on neural dynamics. For full disclosure, we found that the CTM never leveraged this for ImageNet classification (see Section 3), where only 3 of 8196 (i, j) neuron pairs in $\mathbf{S}_{\text{out}}^t$ had any meaningful decay. For navigating 2D mazes (see Section 4) approximately 3% of (i, j) pairs had meaningful decay, indicating that a task with clear sequential reasoning warrants a more local perspective (i.e., a faster decay to synchronization).

Recovering latent snapshot dependency Interestingly, the CTM could learn to recover a dependency on the current state $\mathbf{z}^t$, if we set $i = j$ and r_{ij} is very small, since this would be approximately equivalent to computing the element-wise square of $\mathbf{z}^t$ for each i sampled neuron. In practice this turns out to be unnecessary, but we did explore several subsampling strategies and discuss these in Appendix B.2.

2.5. Loss function: optimizing across internal ticks

The CTM produces outputs at each internal tick, t. A key question arises: how do we optimize the model across this internal temporal dimension? Let $\mathbf{y}^t \in \mathbb{R}^C$ be the prediction vector (e.g., probabilities of classes) at internal tick t, where C is the number of classes. Let y_{true} be the ground truth target. We can compute a loss at each internal tick using a standard loss function, such as cross-entropy:[2]

$$\mathcal{L}^t = \text{CrossEntropy}(\mathbf{y}^t, y_{true}), \tag{11}$$

[2] In practice any appropriate loss function could be used.

and a corresponding certainty measure, C^t. We compute certainty simply as 1 - normalized entropy. We compute $\mathcal{L}^t$ and C^t for all $t \in \{1, \ldots, T\}$, yielding losses and certainties per internal tick, $\mathcal{L} \in \mathbb{R}^T$ and $C \in \mathbb{R}^T$.

A natural question arises: how should we reduce $\mathcal{L}$ into a scalar loss for learning? Our loss function is designed to optimize CTM performance across the internal thought dimension. Instead of relying on a single step (e.g., the last step), which can incentivize the model to only output at that specific step, we dynamically aggregate information from two internal ticks: the point of minimum loss and the point of maximum certainty. This approach offers several advantages: (1) it encourages the CTM to develop meaningful representations and computations across multiple internal ticks; (2) it naturally facilitates a curriculum learning effect, where the model can initially utilize later internal ticks for more complex processing and gradually transition to earlier steps for simpler processing; and (3) it enables the CTM to adapt its computation based on the inherent difficulty of individual data points within a dataset. To this end, we aggregate $\mathcal{L}$ dynamically (per data point) over two internal ticks:

1. the point of minimum loss: $t_1 = \text{argmin}(\mathcal{L})$; and
2. the point of maximum certainty: $t_2 = \text{argmax}(\mathcal{C})$.

The final loss is then computed as:

$$L = \frac{\mathcal{L}^{t_1} + \mathcal{L}^{t_2}}{2}, \tag{12}$$

and stochastic gradient descent is used to optimize the model parameters, θ_{syn} and $\theta_{d=1\ldots D}$ (see Equations 1 and 3).

This approach effectively enables the CTM to improve its 'best' prediction while making sure that high certainty is attributed to correct predictions. Listing 4 shows how the loss is calculated, demonstrating how the certainty is computed as 1 - normalized entropy. This approach also enables the CTM to dynamically adjust its thought process as needs be.

```
def ctm_loss(logits, targets):
    B, C, T = logits.shape
    # B=minibatch size, C=classes, T=thought steps
    # Targets shape: [B]
    # Compute certainties as 1 - normalised entropy
    p = F.softmax(logits, 1)
    log_p = torch.log_softmax(logits, 1)
    entropy = -torch.sum(p * log_p, dim=1)
    max_entropy = torch.log(C)
    certainties = 1 - (entropy / max_entropy)
    # Certainties shape: [B, T]
    # Expand targets over thought steps
    targets_exp = torch.repeat_interleave(targets.unsqueeze(-1), T, -1)
    # Loss function could be other things, but we use cross entropy without reduction
    loss_fn = nn.CrossEntropyLoss(reduction='none')
    # Losses are of shape [B, T]
    losses = loss_fn(predictions, targets_exp)
    # Get indices of lowest loss thought steps for each item in the minibatch
    lowest_idx = losses.argmin(-1)
    # Get indices of most certain steps for each item in the minibatch
    certain_idx = certainties.argmax(-1)
    loss = (losses[:, lowest_idx] + losses[:, certain_idx])/2
    return loss.mean()
```

Listing 4 | CTM loss function, enabling the CTM to be flexible regarding the number of internal ticks used for any given data point.

Introducing timing as a fundamental functional element of the CTM has a number of beneficial properties, one of which is that we can train the CTM **without any restriction on how many internal ticks it should use**. Such freedom, while subtle, is actually quite profound as it lets the CTM attribute variable amounts of compute to different data points. The idea of adaptive/dynamic compute (Graves, 2016) is aligned with modern test-time compute, with the difference being that this sought-after modeling property falls out of the CTM as a consequence, rather than it being applied post-hoc or as a restriction during learning. Note that there is nothing in the loss function that explicitly encourages

this behavior. In some sense, the CTM implements a form of *inductive bias*, where the complexity of the modeling process (approximated by the number of internal ticks used) can be tailored to the data. We believe that this is a far more natural means by which to solve problems of variable difficulty (e.g., easy versus difficult to classify images). That such a property occurs as a consequence of improving biological plausibility is not surprising, but it is gratifying. We contrast the performance and characteristics of the CTM against other models and a human baseline in Section 5.

Experimental evaluation

The following sections present a comprehensive evaluation of the Continuous Thought Machine (CTM) across a diverse suite of challenging tasks. The primary goal of these experiments is to explore the capabilities and characteristics that emerge from the CTM's core design principles: **neuron-level temporal processing** and the use of **neural synchronization as a direct latent representation**. We aim to understand how explicitly modeling and leveraging the unfolding of internal neural activity allows the CTM to approach problems requiring different facets of intelligence.

We begin by examining the CTM on standard perception tasks like ImageNet-1K (Section 3), focusing on analyzing the richness of its internal dynamics, its emergent reasoning processes, calibration properties, and adaptive computation. We supplement these later with studies to compare the CTM to human performance on CIFAR-10 (Section 5) and perform ablation studies on CIFAR-100 (Section 6).

We then specifically probe the CTM's capacity for complex sequential reasoning, planning, and spatial understanding using a challenging 2D maze navigation task designed to necessitate the formation of an internal world model (Section 4).

Further experiments investigate the CTM's ability to learn and execute algorithmic procedures on sequence-based tasks such as sorting real numbers (Section 7) and cumulative parity computation (Section 8), where the temporal unfolding of thought is critical. We also test its capacity for memory, retrieval, and symbolic manipulation through a question-answering task on MNIST digits (Section 9). Finally, we extend the CTM to reinforcement learning environments to demonstrate its applicability to sequential decision-making and continuous interaction with an external world (Section 10).

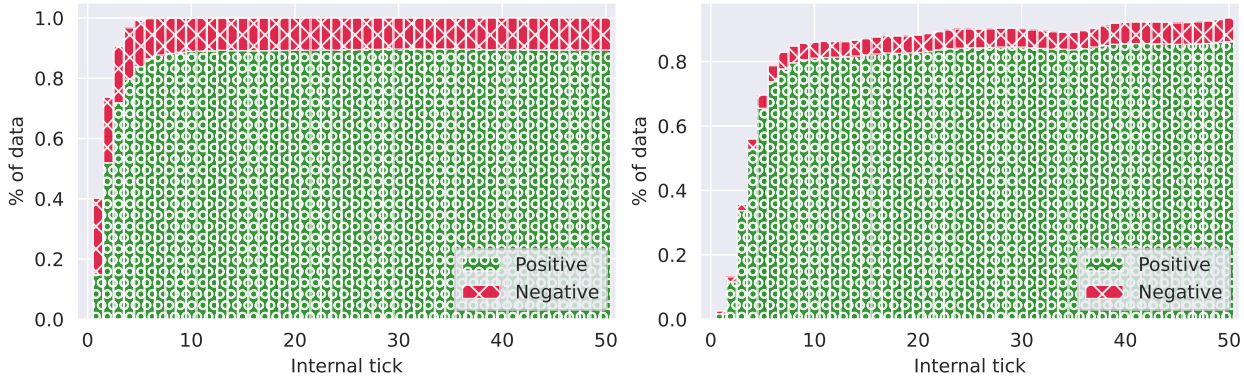
Collectively, these experiments are designed to provide insights into how grounding computation in neural dynamics allows the CTM to develop and utilize internal thought processes, offering a distinct approach compared to conventional models and taking a step towards more biologically plausible artificial intelligence.

3. ImageNet-1K classification

In this section we test the CTM on the ImageNet-1K classification task. We are not claiming that the CTM is state-of-the-art in terms of classification accuracy, which would require a substantial amount of effort and tuning to find an optimal training recipe (Vryniotis and Cord, 2021), but rather that the way in which the CTM solves this task is novel and worth inspection. We detail the model setup and hyperparameters in Appendix C.1. With a ResNet-152 backbone[3], the CTM achieves 72.47% top-1 validation accuracy and 89.89% top-5 validation accuracy, when assessed on uncropped ImageNet-1K validation data (but scaled such that the short side of the image is length 256). While this result is currently not comparable with state-of-the-art techniques, it is also the first attempt to classify ImageNet-1K using neural dynamics as a representation. We anticipate closing this gap that with further advances, more hyperparameter tuning, and feature extractors tailored to the CTM.

[3] altered such that the initial convolution is kernel is 3×3 as opposed to 7×7 to constrain the receptive field – see Appendix C.1 for a discussion thereon

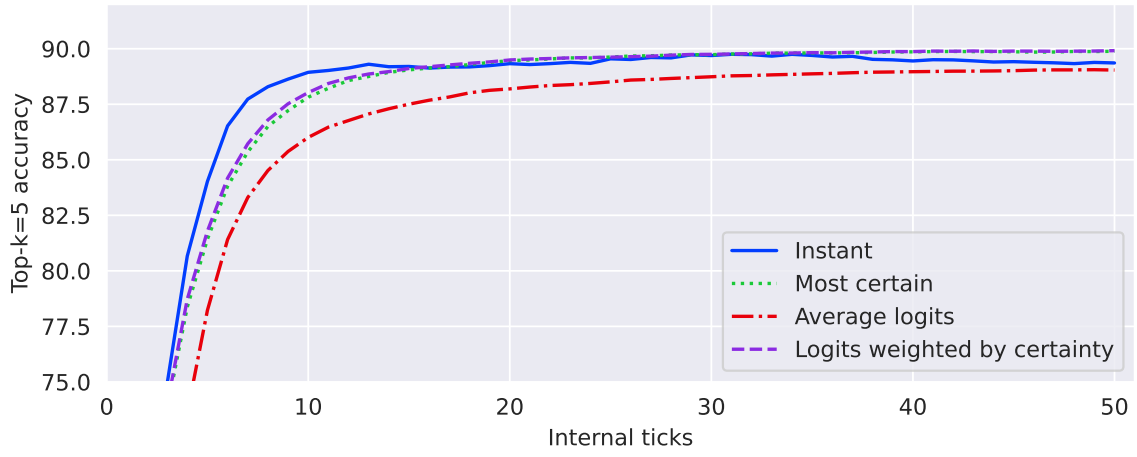
3.1. Prediction analysis: the power of the thought dimension



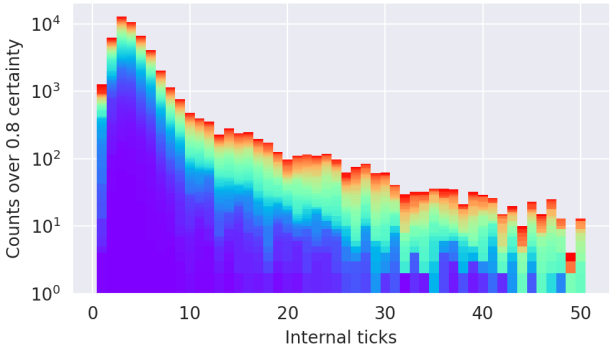
(a) Certainty threshold of 0.5.

(b) Certainty threshold of 0.8.

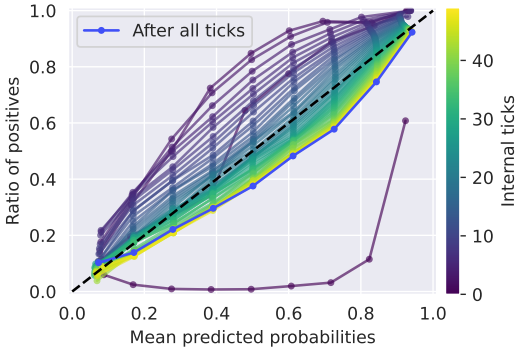
Figure 2 | Top-5 accuracies (validation) only when making a prediction past fixed certainty thresholds. At a lower threshold of (a) 0.5, the CTM will make a prediction 100 % of the time from approximately 4 internal ticks and onward, but (b) will only predict on approximately 80 % of the data when setting a certainty threshold of 0.8. These assessments could be used to tailor a certainty threshold to enable adaptive compute, halting compute as needed.



(a) Accuracy vs. steps for different prediction mechanisms.



(b) Histogram of certainty of at least 0.8.



(c) Calibration plots.

Figure 3 | Exploration of the performance and utility of the CTM, showing the relationship between internal ticks and top-5 ImageNet-1K accuracy. In (a) we show the accuracy versus internal ticks when determining the output prediction in 4 different ways, showing how taking the prediction at a given internal tick is sensible until approximately 15 steps, where it becomes better to consider the certainty as a measure of success. In (b) we show a histogram of data counts exceeding a certainty of 0.8 for each internal tick; color denotes class indices. In (c) we show calibration plots, where predicted probabilities for the CTM are considered to be the average probability up to a given internal tick, showing how this results in good model calibration.

Figure 2 shows how the internal ticks for the CTM could be truncated when a chosen minimum certainty is reached, and what the expected top-5 accuracy would be. For instance, it takes fewer than 20 internal ticks per image to reach a certainty of 0.5 for all data, but when choosing a threshold of 0.8, not all of the data always reaches this threshold. In the latter case, compute could be truncated as needed by the user. By halting the internal ticks when an acceptable internal threshold is met, a form of adaptive compute can be utilized.

Prediction mechanisms: accounting for certainty. Figure 3a shows how different prediction mechanisms can affect overall performance. We show an 'instant' prediction (i.e., the prediction at each internal tick) compared to predictions based on certainty: the maximum certainty up to a given step, and when weighting logits by certainty. Interestingly, after approximately 15 internal ticks it becomes preferable to take explicit account of certainty. Using an unweighted average of logits yields the worst performance, implying that the CTM does indeed undergo a process to improve its prediction, while *potentially moving through incorrect predictions of low certainty*. Figure 5b gives concrete examples of low-certainty instances for further evidence.

Figure 3b shows the distribution of internal ticks when the CTM yields a certainty of at least 0.8, indicating that the majority of the data requires fewer than 10 internal ticks, with a long tail toward a maximum of 50 internal ticks. The calibration plots in Figure 3c are perhaps most striking as they show that the CTM has very good calibration. This is owing to how the CTM becomes increasingly certain over internal ticks: we consider the predicted probability of a given instance to be the average probability over internal ticks of the chosen class. The demonstration in Figure 5 shows how certainty increases over internal ticks. Evidently, following an internal process seems to enable the CTM to produce more trustworthy class probabilities – a characteristic that usually needs post-training adjustments or special training setups (Guo et al., 2017).

3.2. Neural dynamics analysis

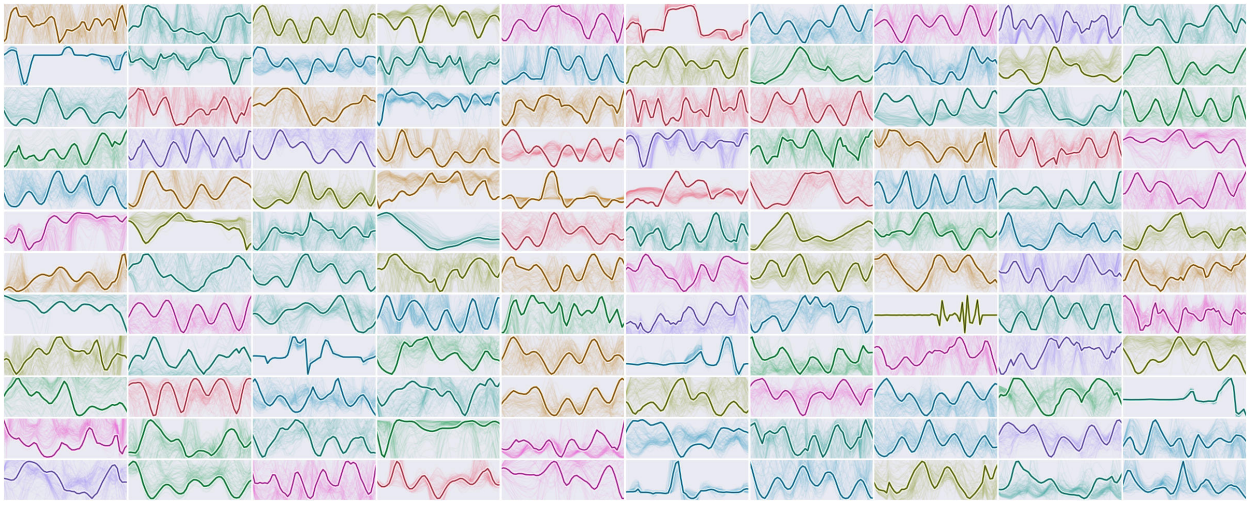


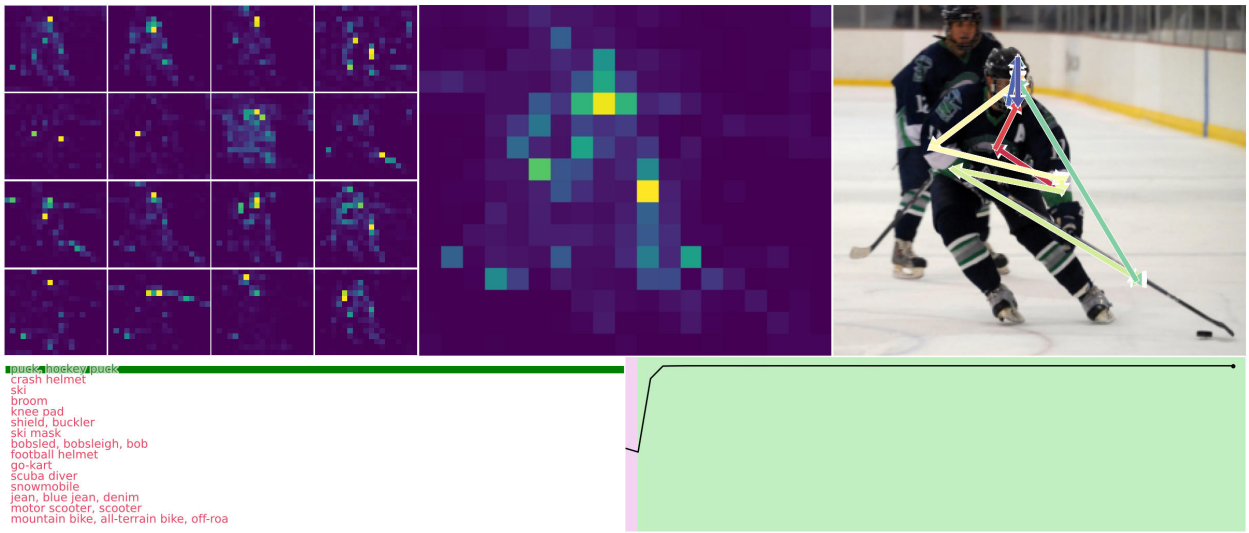
Figure 4 | Post-activation neuron dynamics (⑤ in Figure 1). Each subplot (in a random color) shows the dynamics of a single neuron over internal ticks, where multiple examples from different images are shown as faint background lines and the foreground line is a single example. We show multiple examples as evidence of diversity across data. It is these dynamics that are used when computing synchronization and they form the fundamental processing element of the CTM.

Figure 4 visualizes the post-activation neural dynamics of this CTM. **These dynamics are diverse and rich in structure, and they form the representation with which the CTM takes action and makes decisions.** The purpose of this figure is to show that the CTM does indeed yield a diverse set of neural activities, the dynamics of which can be measured in relationship to one another (i.e.,

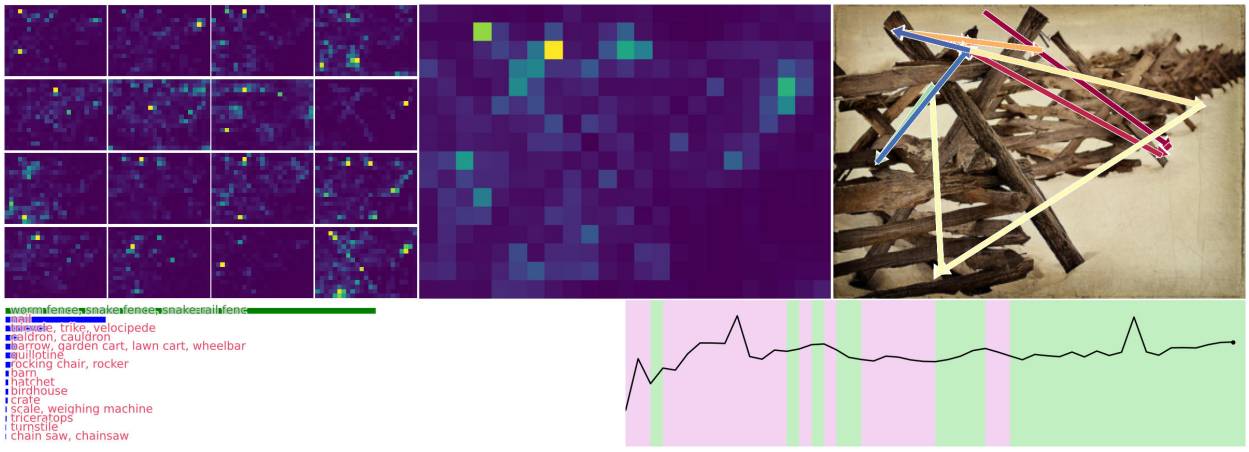
synchronization) and used as a powerful latent representation for downstream tasks. In Section 4 we provide evidence that such a representation has high-utility for problem solving.

Take-home message. Figure 4 shows that the neurons in the CTM exhibit complex multiscale patterns, but have not provided any pragmatic rationale for why this might be useful. The reason we show this is as evidence that we the CTM builds and leverages true **dynamics**, where the patterns of neural activity are non-trivial and diverse. These dynamics and the complexity contained within form a new kind of representation that we believe is closer to the biologically plausible mechanisms behind neural computation.

3.3. Demonstrations: the CTM follows a process



(a) A high-certainty example.



(b) A low certainty example that eventually becomes correct with more internal ticks.

Figure 5 | ImageNet-1K use cases, randomly drawn from the validation set. This demonstration is ideal when viewed as a video as there are 50 internal ticks, but we are showing only the final step. On the left we show the average (over internal ticks) of all 16 attention heads' weightings, and on the right we show an approximation of the center of mass of their collective average (see Appendix C.1 for details) over the full 50 steps as arrows from red to blue. Owing to the sequential nature of the CTM's internal ticks, we observe each head's attention shifting smoothly from region to region, at times zoning into specific salient features (nose, boundaries, etc.), while at other times spreading over wider areas, or even moving in identifiable directions (e.g., from bottom to top). Appendix C.3 contains several additional demonstrations.

Figure 5 shows examples from the ImageNet-1K validation set, as the CTM sees it. We give more examples in Appendix C.3. We encourage the reader to view these visualizations in video form as there

are 50 frames (1 per internal tick) that show how **the attention maps change over time**, shifting to different regions over the CTM's internal thought process. The smooth transition of attention to various parts of the image emerges as a property during training. We attempted to show some of this transitory attention by way of arrows, showing how the attention moves over salient areas an intuitive fashion. Unpacking every interesting facet of these attention map progressions is simply infeasible in this paper. Instead, for these examples we show how the attention patterns demonstrate a complex process. Also shown is the certainty over time, indicating how the CTM becomes more certain as it reasons.

Note that since the CTM uses attention to retrieve information, it is not limited to fixed-size images (and, for future work, can be applied to arbitrarily length token sequences), hence our evaluations on uncropped validation data. One could conceivably build a **hierarchy of tokens** by using multiple input resolutions, letting the CTM attend to a large collection of tokens (during inference, not training), but we reserve this exploration for future work.

The CTM learns to observe over time. During our experimentation we monitored the functionality of the CTM as it progressed through its training. While we do not explicitly show it in this paper, the complexity of the neural dynamics and consequently the complexity of the observation process the CTM undertakes increases during learning. At first the CTM does not 'look around' as it does in Figure 5, only learning that behavior over time. An early work by Xu et al. (2015) demonstrated how to use an RNN to reason over an image for text captioning; the CTM's reasoning process differs in that it unfolds along an internal dimension that is decoupled from both the input and target data, yet it still yields a complex attention pattern that highlights where it focuses its attention when making decisions.

Taking steps toward natural intelligence. Biological intelligence is still superior to AI in many cases (Chollet et al., 2024; Lake et al., 2017; Phan et al., 2025; Ren and Xia, 2024). Biological brains solve tasks very differently to conventional neural networks, which might explain why this is the case. In this work, we aimed to develop a model that approaches problem-solving in a manner more aligned with biological brains, emphasizing the central role of neural dynamics in achieving this similarity. Our observations suggest that the CTM undertakes a process, where it sequentially retrieves information from an image. We show more examples in Appendix C.3, where each instance shows interesting or unique patterns or outcomes.

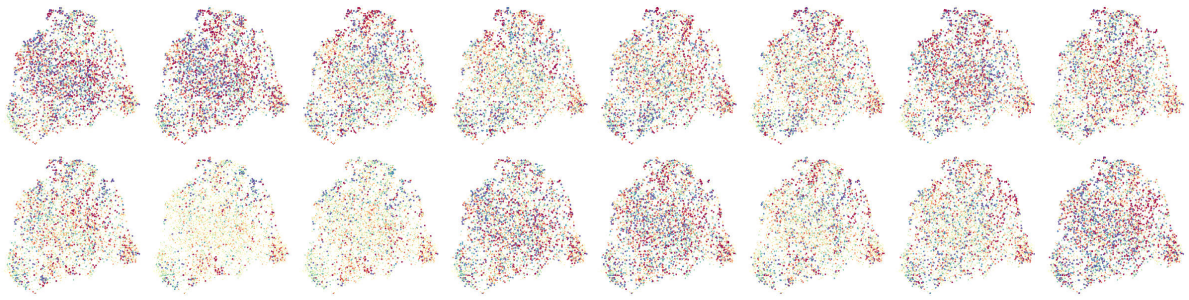


Figure 6 | Observation of the neurons in the CTM as it observes and thinks about an image. Appendix J gives details of how the neurons were arrange using UMAP (McInnes et al., 2018). The colors indicate activations that range from low (blue) to high (red). We show the progression of the neural activity over internal ticks from top left to bottom right. Upon careful inspection one can discover clear structures at multiple scales. This visualization is best viewed in video form.

As a final point of comparison, we consider low-frequency traveling waves, a phenomenon widely documented in cortical dynamics and implicated in various neural computations (Muller et al., 2018). We present Figure 6, where we map the CTM's neurons to a 2D feature space using UMAP (McInnes

et al., 2018). Each neuron's position in this space is determined by its activation 'profile' – its response pattern over both time and multiple stimuli (see Appendix J). Visualizing this mapping over internal ticks reveals low-frequency structures propagating across the feature space (best viewed as a video). Importantly, the CTM generates this structure in an emergent fashion, without any explicit driving signal. Analogous phenomena occur in networks of Kuramoto oscillators (Miyato et al., 2024); in our case, waves propagate across a learned feature map in an all-to-all network. Concurrent work also explores explicitly encoding traveling waves for long-range communication (Jacobs et al., 2025). We do not assign functional meaning to these observed waves but highlight their distinct presence during the CTM's thought process.

4. 2D Mazes: a setup that requires complex sequential reasoning

In this section, we use 2D mazes as a tool to investigate the behavior of the CTM when asked to plan and navigate. Solving a 2D maze can be easy with the right inductive bias: by ensuring the output space matches the dimensions of the input space, where at each pixel a model must perform binary classification. Such a setup is amenable to machines by design, as they can learn iterative algorithmic solutions (Bansal et al., 2022; Schwarzschild et al., 2021), and it excludes the need to think in a more natural fashion. Even so, the trainability of models is questionable, with techniques often relying on careful model and/or objective design that favors generalization to larger mazes (Bansal et al., 2022; Zhang et al., 2025). Such generalization is certainly one important aspect of intelligence.

However, there is a critical distinction between simply finding the solution to a maze, versus **following a thought process to form said solution**. While the emergent behavior of such systems can be impressive (e.g., generalizing to mazes far greater in size (Bansal et al., 2022)), it is difficult to reconcile whether these models are demonstrating intelligence. How can we make the 2D maze task more challenging, such that a human-like solution is required? We propose the following:

1. **Constrain the output space directly** as a set of steps from start (denoted by a red pixel) to finish (green pixel). We require a solution in the form of a fixed-sized array (length 100[4]) containing 1 of 5 movement types (**L**eft, **R**ight, **U**p, **D**own, or **W**ait[5]) per step. This mitigates against the types of simple solutions described above and requires more of an understanding of the target maze.
2. **Disallow positional embeddings** when using attention. There are two reasons for this: (1) it forces the model to build an internal 'world representation', where it can only craft attention queries based on its ongoing understanding of the data; and (2) it enables seamless scaling to bigger maze images (see Section 4.3).

It is our hope that this new phrasing of the 2D maze task provides a challenging benchmark that can highlight models that are able to follow a thought process. We used the maze-dataset repository (Ivanitskiy et al., 2023) to generate 39×39 mazes for training and 99×99 for generalization tests[6]. We trained three model variants for comparison:

1. A **CTM** with a constrained ResNet-34 backbone, using the first two hyper-blocks only. This CTM is nearly identical in structure to those used in image classification (Sections 3). At each internal tick the CTM outputs a matrix that defines a route from the start location, $\mathbf{y}^t \in \mathbb{R}^{100\times5}$ (see Equation 6). It is trained with the variable-internal tick loss (Equation 12). We also adjust the loss function to use a curriculum approach, optimizing for early steps in the route over later steps. Appendix D.2 details hyper-parameters and Appendix D.3 explains the curriculum approach.

[4]This may be shorter than required for some mazes, in which case we ignore later steps.
[5]We use 'wait' classes for instances where the route is shorter than 100, and fill the target vector with wait classes
[6]See the CTM code repository

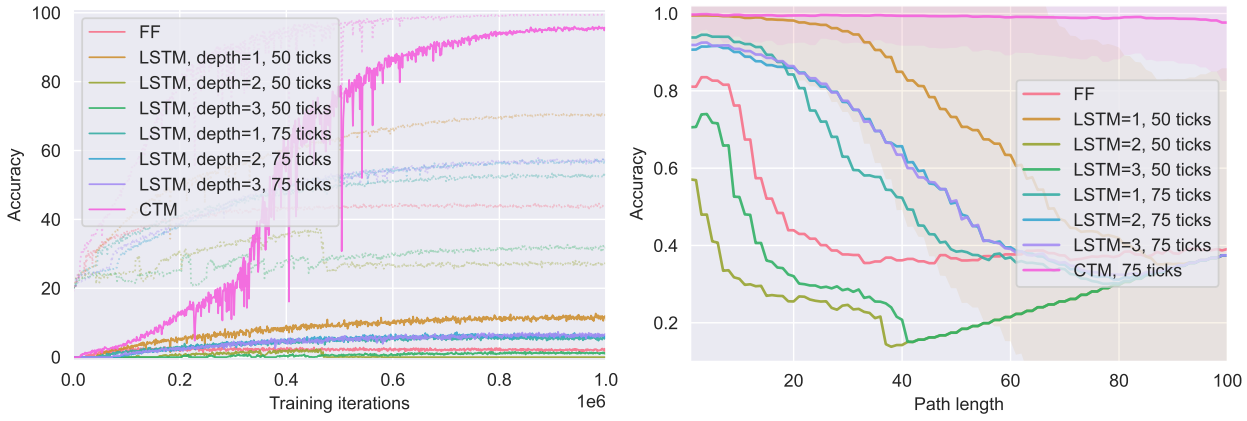
2. 1, 2, and 3-layer **LSTM baselines** using the same model width as the CTM (see Appendix D.4 for details). The LSTM baseline also uses a curriculum approach, but was unable to learn beyond a small number of steps in the maze.
3. A **feed-forward-only** (i.e., no recurrence) model (FF) where the features were projected via a hidden layer (same width as the CTM) to the prediction (see Appendix D.4 for details). Since we use no positional embedding for the CTM and LSTM models they must learn to build an internal representation of the data they are seeing, but no such mechanism is available to the FF model. Therefore, we flatten the final ResNet features and project those to $\mathbf{y}^t$ instead as this lets the FF model learn spatial context directly.

Using the same hidden width, the CTM required the fewest parameters. See Appendix D.4 for more details.

4.1. Results

Figure 7a shows the accuracies of the CTM versus baselines. The FF model and the best LSTM model both show signs of overfitting (see Appendix D.5 for loss curves), indicating that their structure is poorly suited to the problem. Only the CTM achieves high accuracy on this task. During our experimentation we simply could not get the LSTM to achieve the same performance, with the single-layer LSTM using 50 internal ticks achieving the best performance. Upon inspection of the solution, and as shown by the orange curve ("LSTM=1, 50 ticks") in Figure 7b, we can see that the LSTM is beginning to learn a solution, but is unable to push beyond that.

Trainability. The large disparity in performance between the CTM and LSTM in this case raises questions of trainability, with the CTM being far easier to optimize. Solving the maze task is complex because it requires a model to create and complex representation that modulates data interaction, produces a route prediction, and also maintains a memory of its positioning thus far (see Section 4.4 for a longer discussion). The fact that the CTM can do this with minimal modifications (only the form of the predictions) is testament to its utility.



(a) Accuracies, route (solid) and per step (dotted).

(b) Accuracies versus path length..

Figure 7 | CTM versus a feed-forward baseline and several LSTM setups. The CTM is the only model that can fit sufficiently to the training data (nearly perfect train accuracy), does not overfit (indicating a good choice of inductive bias (Utgoff, 2012)), and achieves high test accuracy for longer paths.

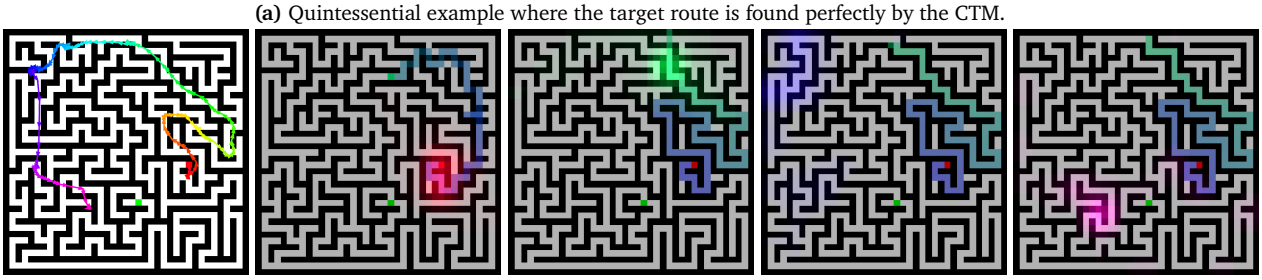
Figure 7b shows accuracies versus route length on the held-out test set. The CTM is clearly more capable of solving longer mazes, while the baseline methods start failing early on, with the best performing LSTM losing capability after approximately 20 steps along maze paths. What this indicates is that the CTM is more capable of learning to solve difficult problems. The depth 1 LSTMs were the

closest in terms of parameter counts, but all baselines had more parameters. In other words, the CTM is not performing better because it has more parameters, but rather because of the ideas it is based upon: **neural dynamics and synchronization are useful**.

4.2. Demonstrations: the CTM learns the general procedure

Figure 8 shows the process followed by the CTM. By visualizing the average (across heads) attention weights over time, we can see how the CTM methodically steps along a plausible path until it reaches what it predicts to be the end of the maze. This problem-solving process is *not quite entirely unlike* how a human might approach solving a maze from the top down. We remind the reader to note now that this maze-solving CTM is not using any positional embedding, meaning that in order for it to follow a path through the maze it must craft the cross-attention query by 'imagining' the future state of the maze: a process known as 'episodic future thinking' (Atance and O'Neill, 2001) in humans.

(a) Quintessential example where the target route is found perfectly by the CTM.



(b) An example of **emergent behavior**, where the CTM attends along the path beyond its training steps.

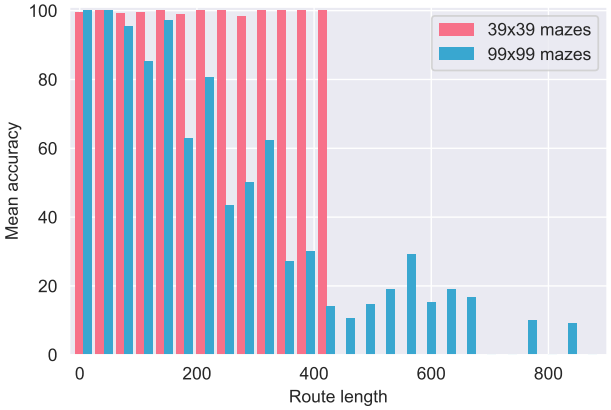
Figure 8 | A natural maze solving approach. Each row shows how the CTM solves a different 39×39 maze. The leftmost images show, as colored arrows, the center of mass of attention throughout the thought processes, demonstrating how the CTM attends along the solution route. The images to the right are snapshots of the solution the CTM is outputting at different internal ticks, overlayed with an attention heatmap (color matched to the arrows). (a) shows a quintessential example, and (b) shows how the CTM continues to attend along the route even beyond the internal ticks used during training (for this we let the CTM unfold $2\times$ longer than it was trained for). We give additional examples and provide an **interactive demonstration where you can interact with the CTM** to solve mazes like this on our project page.

This CTM was trained to predict up to **100 steps outward from the start location**. In Figure 8a we can see how the CTM fixates its attention on the end of the maze (rightmost green heatmap; approximately 75 internal ticks) since this is within the 100 steps it can predict. Contrast this with Figure 8b, where the ground-truth path is far longer than 100 steps. The attention pattern **continues to trace out the remainder of the path** when we assess the attention maps using more internal ticks than what was used during training: to produce these visualizations we ran the CTM for $2\times$ the internal ticks it was trained on.

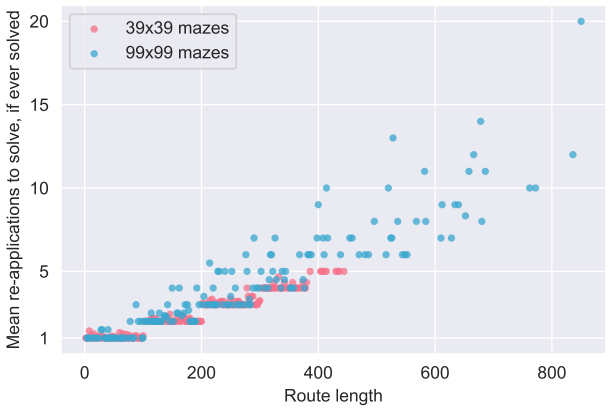
This behavior is emergent and suggests that the CTM has learned a general procedure for the underlying maze task, as opposed to merely memorizing the training data.

In the following section we demonstrate how this CTM can generalize to mazes bigger than what it was trained on.

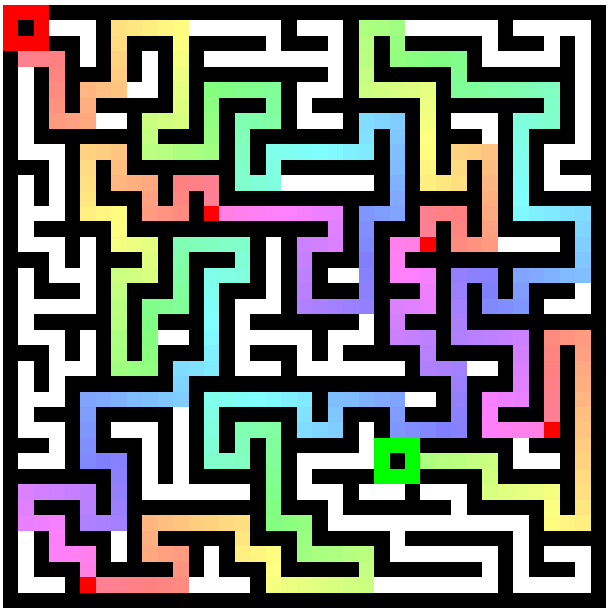
4.3. Generalizing to longer paths and bigger mazes



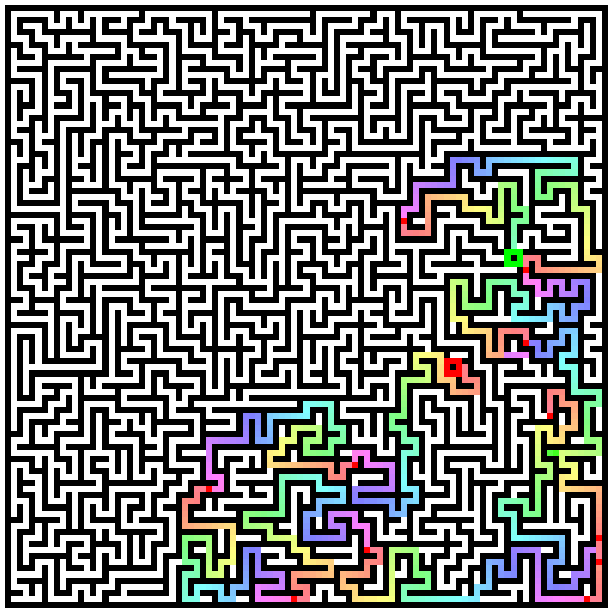
(a) Accuracies when generalizing



(b) Mean re-application count.



(c) Generalizing, 39×39



(d) Generalizing, 99×99

Figure 9 | CTM accuracy when generalizing to longer paths and bigger mazes. This CTM was trained to solve mazes of size 39×39, up to a route length of 100 (truncated for longer routes in the training data). In (b) 're-application' occurs when we move the start point to the end of where the CTM predicts for a given maze. We show the generalization examples for (c) 39×39 and (d) 99×99 mazes, where the rainbow of colors (from red to blue) denote the predicted steps per re-application. The initial start and end points are made larger for viewing clarity.

Our observations in the previous section suggested that the CTM might generalize beyond the data it was trained on. One of the reasons we decided to forgo any positional embedding was that such a model could be applied to a maze of any size without changes. To test this we applied the CTM to longer paths and bigger mazes. We set this up as follows:

1. To test for **longer paths** we applied the CTM to the same-sized mazes as were used in training (39×39), but re-applied the CTM whenever a maze was encountered that had a path longer than 100 (that which the CTM was trained to output). For re-application we simply moved the starting point (red pixel) to the **final valid position when following the path output** by the CTM at its final internal tick (of 75).
2. When **generalizing to bigger mazes**, we follow the same protocol as longer paths, but test on mazes of size 99×99.

Figure 9 shows the results when generalizing to longer routes and bigger mazes. The CTM performs nearly perfectly for any length route on the 39×39 mazes, but performance begins to taper off for the larger 99×99 mazes. This is probably owing to the larger absolute distances between start and end points for larger mazes. For future work, we expect to explore a continuous training regime that: (1) accounts for the end point predicted by the CTM, (2) keeps the current neural dynamics, (3) 'teleports' the starting point to the predicted end, and (4) continues from there for the next minibatch. Such a setup would be better suited to the sequential nature of the CTM. We encourage readers to use our **interactive demonstration where you can interact with the CTM** to solve mazes like this on our project page. See the future work discussion in Section 12 for a discussion on 'open-world' training.

4.4. Discussion: the need for a world model and cognitive map

Internal models of the world and cognitive maps represent crucial aspects of intelligent systems (Gornet and Thomson, 2024; Ha and Schmidhuber, 2018; LeCun, 2022). In this case, we consider a world model to be an internal representation of the external environment, encapsulating an agent's knowledge about the world's structure, its dynamics, and its actionable place therein. A good world model should enable an agent to reason about the world, plan, and predict the consequence of its actions. Cognitive maps (Gornet and Thomson, 2024) specifically focus on spatial relationships and navigation. The ability to construct and utilize these internal representations is a strong indicator, and arguably a prerequisite, for sophisticated intelligence. The notion of 'episodic future thinking' (Atance and O'Neill, 2001) is even considered a hallmark feature of human intelligence. An agent devoid of a world model would be limited to reactive behaviors. Similarly, lacking a cognitive map would severely restrict an agent's ability to navigate and interact effectively within complex spatial environments. Therefore, the presence and sophistication of world models and cognitive maps can serve as a benchmark for evaluating intelligence.

To this end, we designed the maze task such that it would require a good internal world model to solve. This was achieved by (1) requiring the model to output a route directly, as opposed to solving the maze with a local algorithm (Schwarzschild et al., 2021), and (2) forgoing any positional embedding in the image representation, meaning that the model must build its own spatial cognitive map in order to solve the task (Gornet and Thomson, 2024). Indeed, we saw that the NLMs and synchronization components of the CTM enables it to solve our 2D maze task, far surpassing the best baselines we trained. These results suggest that the CTM is more capable of building and utilizing an internal model of its environment.

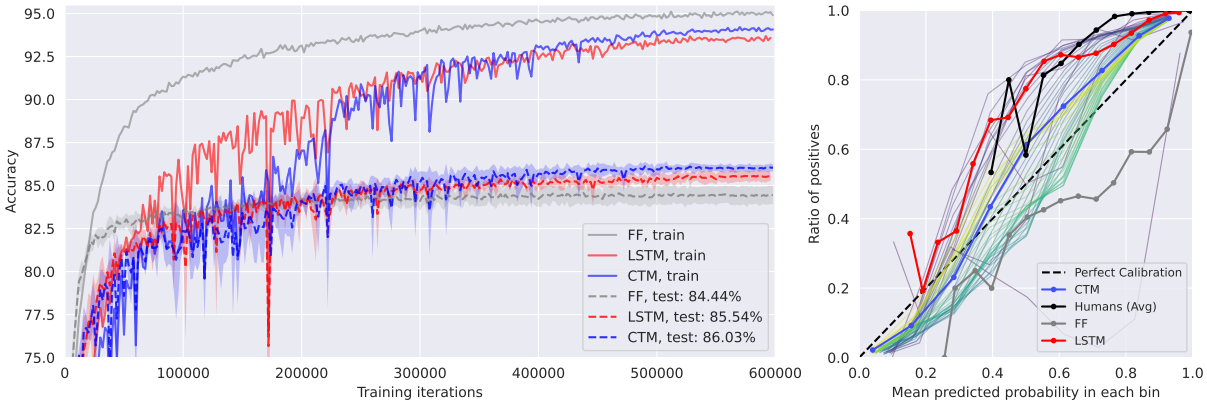
5. CIFAR-10: the CTM versus humans and baselines

In this section we test the CTM using CIFAR-10, comparing it to human performance, a feed-forward (FF) baseline, and an LSTM baseline. For the model-based baselines, we used a constrained featurization backbone in order to emphasize the differences owing to the model structure post-featurization (i.e., CTM versus LSTM versus FF). We also used 50 internal ticks to give the CTM and LSTM 'time to think'. We give full architecture details in Appendix E. The human and model baselines were set up as follows:

- **Human baseline**. We used two datasets of human labels for CIFAR-10; we call these CIFAR-10D (Ho-Phuoc, 2018) owing to its calibration of difficulty levels, and CIFAR-10H (Peterson et al., 2019) originally used to quantify human uncertainty.[7] We used CIFAR-10D to determine easy versus difficult samples, and CIFAR-10H as a direct human baseline.

[7] CIFAR-10D can be found at https://sites.google.com/site/hophuoctien/projects/virec/cifar10-classification; CIFAR-10H can be found at https://github.com/jcpeterson/cifar-10h

- **FF baseline**. A feed-forward only baseline (denoted FF). An MLP was applied to ResNet features after average pooling, where the width of the hidden layer was set to match the parameter count of the CTM for this experiment.
- **LSTM baseline**. An LSTM set up to unroll with an internal thought dimension, with a hidden width set to match the parameter count of the CTM. The LSTM could attend to the image at each step and used the same loss as the CTM for valid comparison.



(a) Training curves. **(b)** Calibration plot.

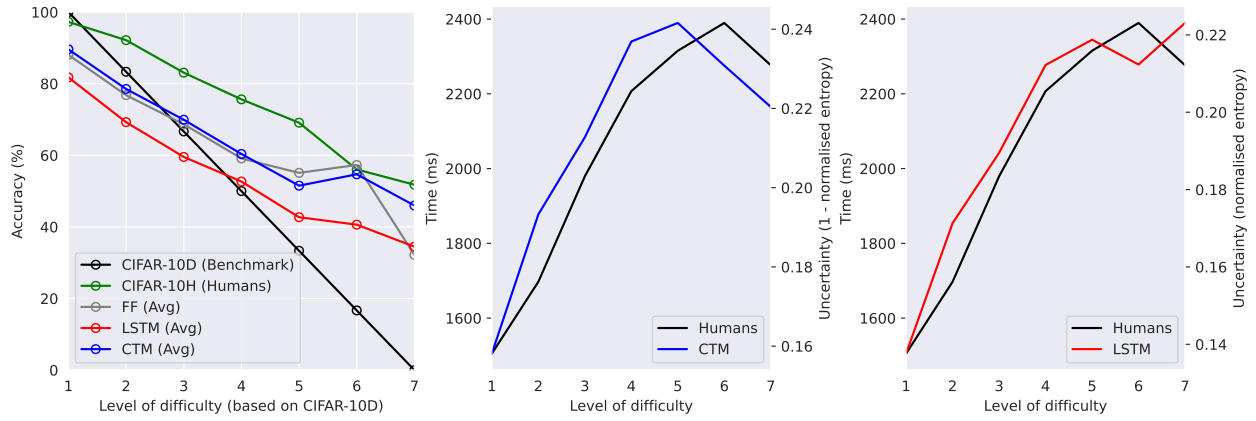
Figure 10 | CIFAR-10 training curves (average over 3 seeds) and calibration plots for the CTM, a feed-forward only baseline, and an LSTM baseline. The CTM is slower than the LSTM per forward pass (±2.4×) but is also more stable during learning. The CTM has the best test performance. The calibration plot shows that even a human baseline (Peterson et al., 2019) is poorly calibrated, and that the CTM demonstrates good calibration, failing in a way that is strikingly similar to humans.

Figure 10 shows the training curves of the CTM, FF, and LSTM models, and calibration plots for each, including an estimation of human calibration using CIFAR-10H. The FF baseline reaches a high training accuracy early on, but also demonstrates a poor generalization gap. The LSTM is less stable during training (we had to set the learning rate to 0.0001 for all experiments because of this) and yields a marginally improved test accuracy. The CTM is more stable and performant.

For the human calibration we used the probabilities provided in CIFAR-10H, which were computed using guesses from multiple humans. We computed calibration here as we did for ImageNet-1K (see Figure 3c): we compute the predictive probability as the average probability for the chosen class over all internal ticks. None of the models are perfectly calibrated, but the CTM demonstrates the best calibration, even when compared to humans. Strikingly, the CTM has even better calibration than humans, while the LSTM follows the human under-confidence.

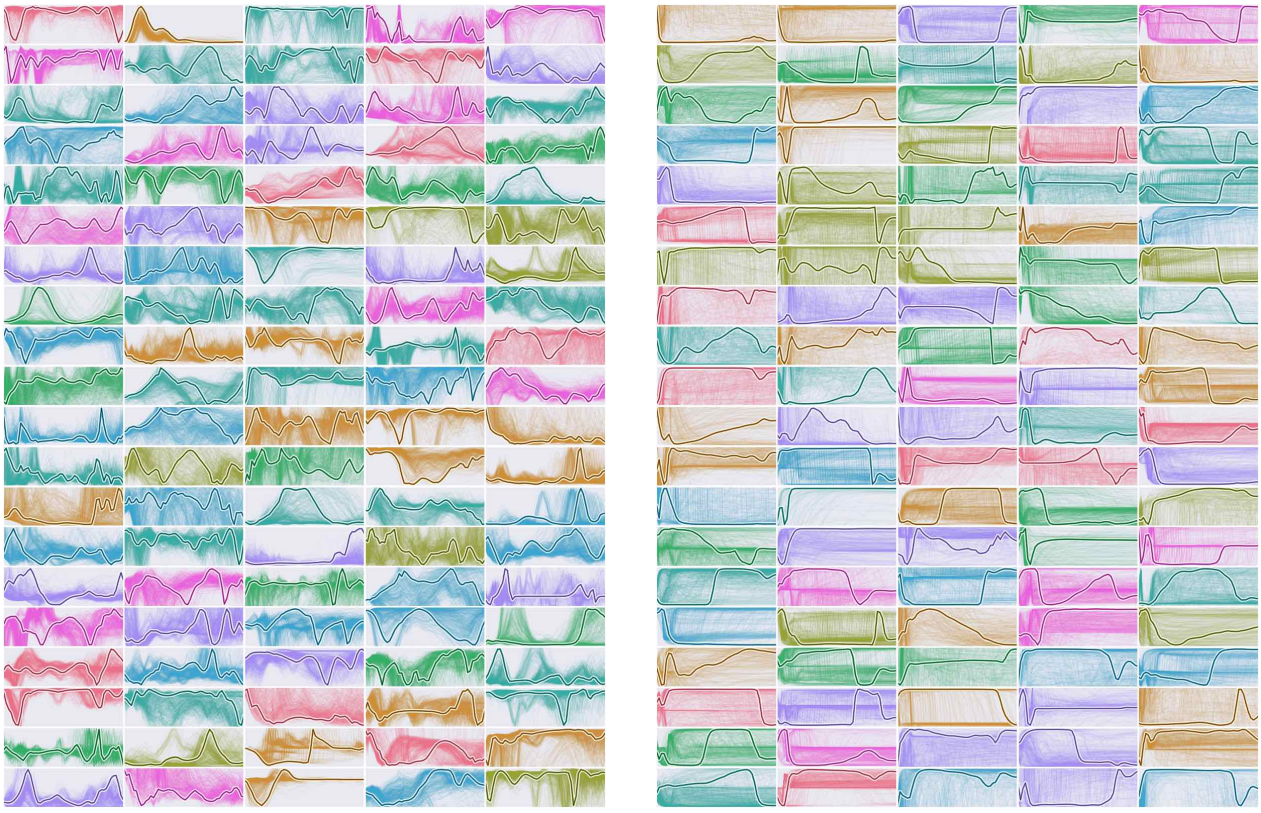
Figure 11a compares models and CIFAR-10H against the difficulty determined using the CIFAR-10D dataset. Each model and humans have similar trends in this case, although the CTM follows most closely to CIFAR-10H. Figures 11b and 11c compare the uncertainties of the CTM and LSTM to the uncertainties of humans (using reaction times from CIFAR-10H as a proxy for uncertainty). We compute the CTM and LSTM uncertainties using the normalized entropies (see Section 2.5) averaged over internal ticks as this approximates the total uncertainty each model has regarding the observed data. Both the CTM and LSTM exhibit trends similar to human reaction times.

Figure 12 shows the neural activities for the CTM and the LSTM baseline. The CTM yields rich, diverse, and complex dynamics with multiple interesting features, including periodic behavior (there is *no periodic driving function*). The distinct difference between the CTM and LSTM neural activities is evidence that the two novel elements of the CTM (NLMs and synchronization as a representation) enable neural dynamics as a fundamental computational tool.



(a) Performance vs. difficulty. **(b)** CTM uncertainty vs. difficulty. **(c)** LSTM uncertainty vs. difficulty.

Figure 11 | Analysis of model and human performance versus difficulty. We used the difficulty calibration from Ho-Phuoc (2018) and compared human predictions from CIFAR-10H (Peterson et al., 2019). We assume that human reaction times are a reasonable proxy for uncertainty and compare this to the trend in uncertainty for the CTM and a parameter-matched LSTM baseline. The error visualized here is a scaled standard deviation.



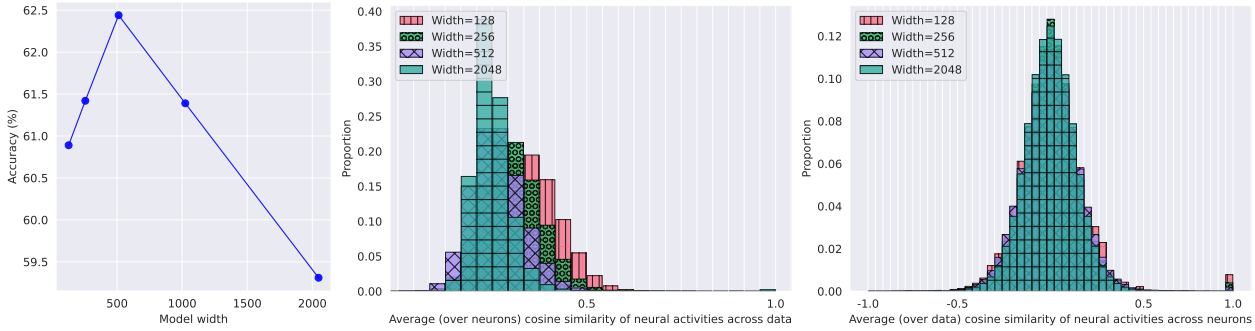
(a) CTM neural activity. **(b)** LSTM neural activity.

Figure 12 | Neuron traces for the CTM and an LSTM baseline, showing how the CTM produces and uses complex neural dynamics even when classifying CIFAR-10. The LSTM yields some dynamic behavior in the post-activation histories shown here, but not nearly to the same degree. Each subplot (in a random color) shows the activity of a single neuron over internal ticks, where multiple examples for different images are shown as faint background lines, and the foreground line is from a randomly chosen example.

6. CIFAR-100: ablation analysis

In this section we explore two aspects of the CTM: (1) width (i.e., number of neurons), and (2) number of internal ticks. We used CIFAR-100 in the experiments discussed below as it is a more challenging dataset than CIFAR-10, while remaining relatively low-demand regarding compute.

6.1. Varying the number of neurons



(a) CIFAR-100 accuracies. **(b)** Neuron similarity across data. **(c)** Neuron similarity across neurons.

Figure 13 | CIFAR-100 accuracies and neuron similarities for different model widths. For (b) neuron similarity across data, we computed the average (over neurons) cosine similarities between matched neurons for all pairings across a sample of 128 images – each bar is the proportion of image pairings that have this average neuron similarity. For (c) neuron similarity across neurons, we compute the average (over data) cosine similarities for all pairs of neurons within each model – each bar is the proportion of neurons having that average cosine similarity. Cosine similarity absolutely closer to zero indicates dissimilarity, and hence improved neuron diversity.

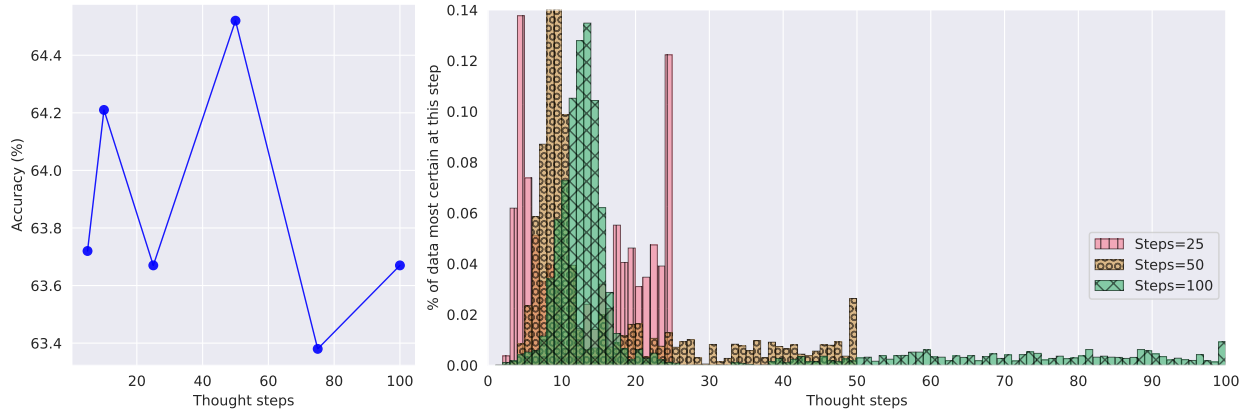
Figure 13a shows CIFAR-100 accuracy versus model width (i.e., the number of neurons) for a fixed backbone network (details of which in Appendix F.1), evidencing improved test performance to a point, and then a reduction in performance. The performance drop-off might be related to overfitting, but it might also be that a wider model requires more training (we set a fixed number of training iterations).

Figures 13b and 13c show a relationship between model width and the diversity of neural activity. Intuitively, we expect that with more neurons we would observe a greater degree of neural activity, and these distributions show exactly that. In Figure 13b we see that when measuring cosine similarity on a neuron-level across data points (averaged over all neurons), a wider model results in a tighter distribution around zero. This means that a wider model results in less similar neurons, indicating that the CTM can encode more information about a data point in its neural dynamics when there are more neurons to work with. Figure 13c shows a similar quantity, where we measure the cosine similarity across neurons for the same data points (averaged over many different data points). In this case the wider model only results in a slightly tighter distribution.

6.2. The impact of longer thinking

Figure 14 explores the impact of internal ticks on the CTM, showing (a) accuracies versus internal ticks and (b) the distributions over internal ticks where the CTM is most certain. The accuracies in Figure 14a are close, although the CTM using 50 internal ticks was the most performant. This suggests once more that with more internal ticks more training might be warranted.

The emergence of two regions of high certainty in Figure 14b is interesting as it indicates that these CTMs do indeed benefit from having more ‘time to think’, perhaps following two different processes internally depending on the data. Although it is difficult to say exactly why this emerges, the fact that these distributions are far from uniform indicates a more complex process than simply computing the result in a strictly feed-forward nature; more analysis is required in future work.



(a) CIFAR-100 accuracies.

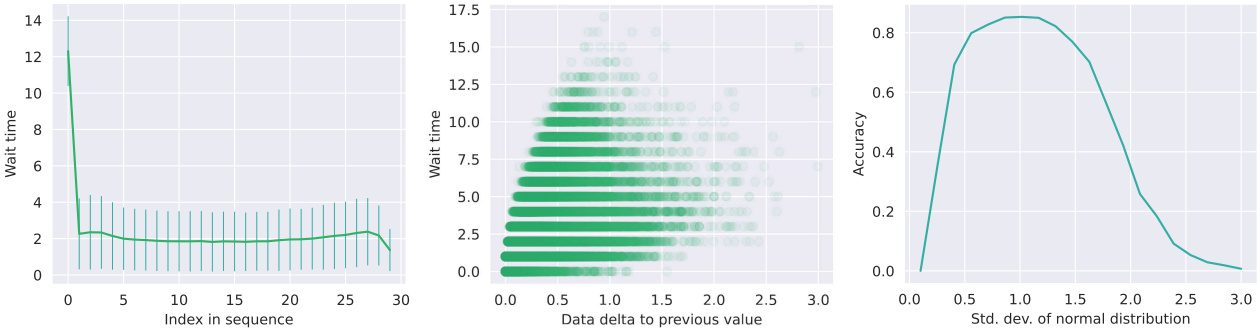
(b) Distribution of where the CTM is most certain.

Figure 14 | CIFAR-100 accuracies and internal tick analysis. The distributions and accuracies in (b) are computed for those internal ticks (x-axis) where the CTM's were the most certain (see Section 2.5). In each case the CTM has two regions of certainty, early on and later, regardless of how many internal ticks are used.

7. Sorting

In this section, we apply the CTM to the task of sorting 30 numbers drawn from the normal distribution. Sorting real numbers was a task explored by Graves (2016) when designing RNNs for adaptive compute, and it provides a test bed for understanding the role of compute for an adaptive-compute system, such as the CTM. In this case the CTM does not use attention, but rather ingests the randomly shuffled input data (30 real numbers) directly. This is implemented by replacing the attention mechanism with a straightforward concatenation, replacing (10) in Figure 1.

Can the CTM produce sequential outputs? For this experiment we set the CTM up to output a sequence over its internal ticks. This is a more standard approach to modeling sequences and we wanted to understand whether the CTM could be trained in this fashion. At each internal tick the CTM output a vector of length 31, including 30 indices for sorting and the 'blank' token used for the well-known connectionist temporal classification (CTC) loss (Graves et al., 2006). We then applied this CTC loss over the full output of the CTM over its internal ticks.



(a) Mean wait times per sequence index (measured in ticks).

(b) Wait times versus change in sequence values.

(c) Generalizing beyond the distribution used for training.

Figure 15 | Results when sorting on $\mathcal{N}(0, I_{30})$. In (a) we can see an evident pattern in the average wait times, where the initial wait time (number of internal ticks) is high, goes to its lowest point, and has a slightly higher bump toward the end of the sequence. In (b) we see that the CTM employs various wait times, but that the difference between the previous output value and the current output value ('data delta') impacts wait time. In (c) we see how this CTM can scale to data drawn from different normal distributions.

Figure 15 gives the results of the CTM on the sorting task. There is a clear pattern to the process it follows, as evidenced by a correlation between wait times and both the current sequence index (a) and the difference between the previous value and the current value being output (b). A similar task was explored by Graves (2016), who sorted 15 numbers using an adaptive compute RNN. In their case, they observed similar wait times before beginning output (analogous to our first sequence element) and also near the end of the sequence. Our analysis of the relationship between wait times and the difference between current and previous data values (what we call 'data delta' in Figure 15b constitutes evidence that the CTM is using an internal algorithm that depends on the layout of the data. We also show that this CTM generalizes to distributions outside of the training data.

Figure 16 demonstrates the CTM's wait times in a real use-case. The red bars indicate longer than average wait times for a given index, and green bars indicate shorter than average wait times. Longer wait times tend to be related to bigger gaps between data points ('data delta' in Figure 15b).

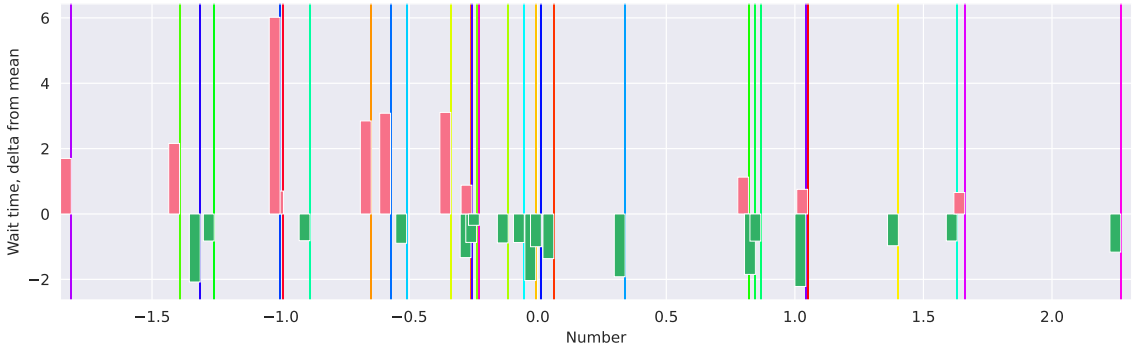
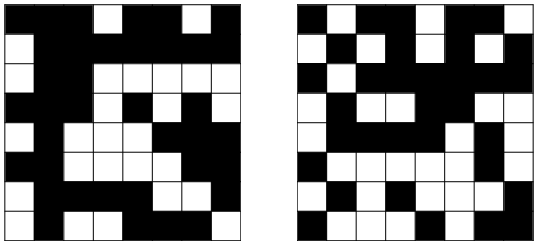


Figure 16 | Sorting demonstration. The input data is represented as vertical lines whose colors denote their original shuffled position (from purple through to red in the 'rainbow' colormap). The red and green bars show positive and negative deviation from the mean wait time (Figure 15a for each index in the sequence), respectively.

8. Parity



(a) Input. **(b)** Target.

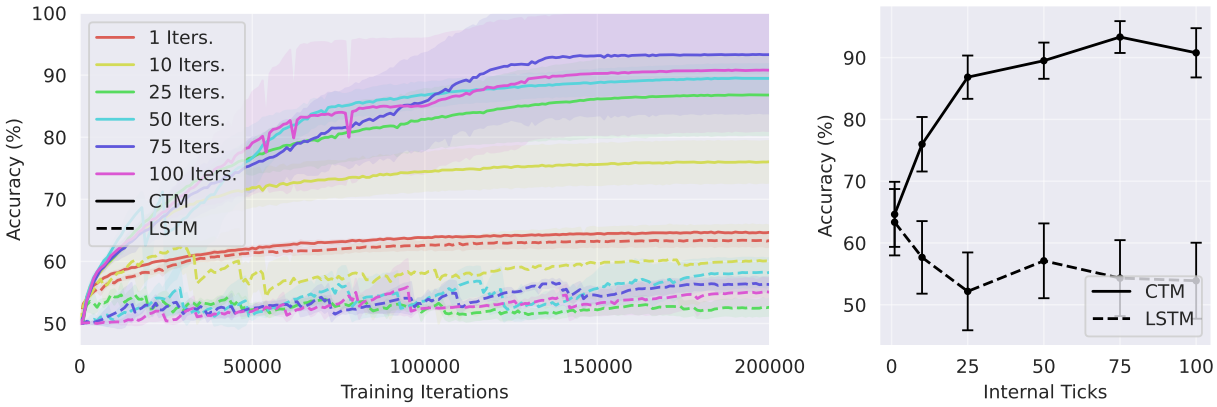
Figure 17 | Parity task. The input (a) is a sequence of 64 binary values (top left to bottom right), and the target (b) is the cumulative parity at each position. Here □ indicates positive parity and ■ indicates negative parity.

The parity of a binary sequence is given by the sign of the product of its elements. When processing a sequence element by element, an RNN could conceivably compute parity by maintaining an internal state, flipping an internal 'switch' whenever a negative number is encountered. However, if the entire sequence is provided simultaneously, the task increases in difficulty due to the increasing number of distinct patterns in the input. Previous work (Graves, 2016) has addressed this challenge using recurrent models, which can learn sequential algorithms for statically presented data. Computing parity, as posed in this manner, is well-suited for testing the capabilities of the CTM.

We apply the CTM to the task of computing the parity of a 64-length sequence containing the values 1 and -1 at random positions. Unlike Graves (2016), we set up the task such that the model computes

the cumulative parity at every index of the sequence, not just the final parity. An example is shown in Figure 17. The values -1 and 1 are embedded as learnable vectors combined with positional embeddings, using attention to ingest input data. We train the CTM with the loss function described in Section 2.5. As a baseline we also trained an LSTM, but set t_2 to be the final iteration since this gave the best results and stability for LSTM training. See Appendix G for more details.

8.1. Results

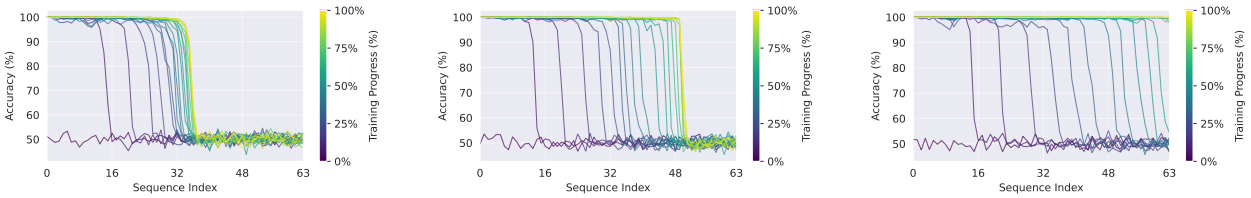


(a) Training curve for CTM and LSTM configurations.

(b) Accuracy vs internal ticks.

Figure 18 | The training curve (left) and the final accuracy vs internal ticks (right) for various CTM and LSTM configurations. The shaded areas and error bars represent one standard deviation across seeds. For the CTM, increased thinking time leads to an increase in performance.

Accuracy increases with thinking time. Figures 18a and 18b show the training curves and final accuracies for various configurations of the CTM, where we changed the number of internal ticks (T) and memory length (M). We also plot parameter-matched LSTM baselines for comparison. Generally, the accuracy of the CTM improves as the number of internal ticks increases. The best-performing models were CTMs with 75 or 100 internal ticks, which could reach 100% accuracy in some seeded runs. The LSTM baselines, on the other hand, struggle to learn the task, with the best performing LSTM, which had 10 internal ticks, achieving an accuracy of 67% ± 0.05%. The LSTM baselines with over 10 internal ticks demonstrate unstable learning behavior; this mimics our observations in Section 4.1, that simple recurrent models are not necessarily well-suited to unfolding an internal thought process. Although the CTM exhibits much more stable training, there is considerable variance in final accuracies due to the choice of random seed. This is discussed in more detail in Appendix G.4.



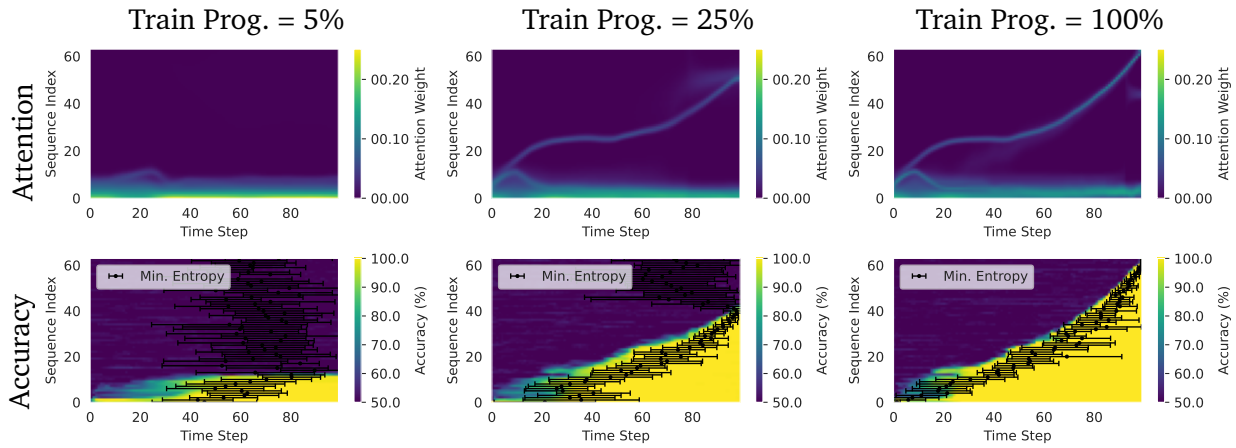
(a) 10 internal tick CTM.

(b) 25 internal tick CTM.

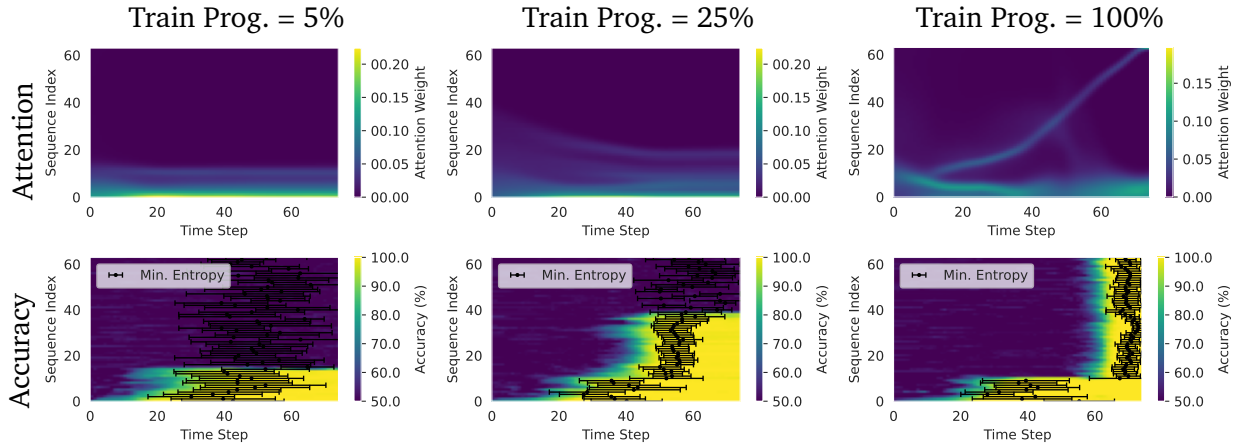
(c) 75 internal tick CTM.

Figure 19 | Accuracy across the 64-element sequence at different training stages (indicated by color) for various internal tick configurations. Early in training, all CTMs accurately predict parity only for initial sequence elements, gradually improving for later elements as training progresses. Models with more internal ticks achieve higher accuracy, with the 10-step model (a) correctly predicting approximately half the sequence and the 75-step model (c) correctly predicting the entire cumulative parity sequence.

The CTM learns a sequential algorithm. To analyze how the CTM learns to solve the parity task, Figure 19 shows the accuracy for each of the 64 elements in the input sequence at different stages of training, for three different internal tick configurations. The models first learn to predict the parity of the initial elements, and as training proceeds, learn to predict later and later positions. With more internal ticks, the model can accurately predict more elements in the target sequence.



(a) CTM trained with 100 internal ticks.



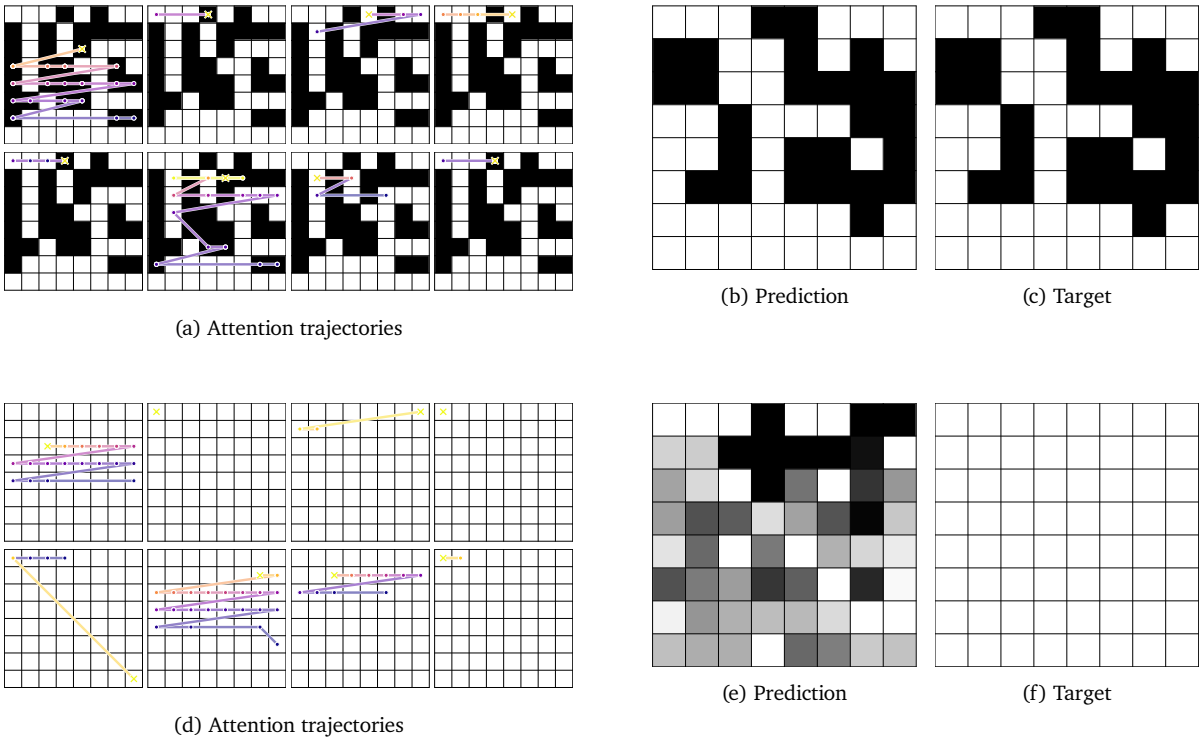
(b) CTM trained with 75 internal ticks.

Figure 20 | Attention patterns (top) and accuracy (bottom) at different points in training, for a CTM trained with 100 internal ticks (a) and 75 internal ticks (b). The black points in the accuracy plots denote the internal tick at which the model reached maximum certainty, with the error bars denoting one standard deviation across samples.

To gain insight into how the model solves the cumulative parity task, we visualize the CTM's attention patterns, accuracy, and points of highest certainty across all 64 elements at multiple stages of training in Figure 20 for two different models. The attention and certainty patterns evidence that these CTMs are leveraging different algorithms to solve the cumulative parity task. When using 100 internal ticks, attention moves from the beginning to the end of the sequence, and with it, the model increases its certainty of the prediction at that position. The CTM with 75 iterations, on the other hand, learns to attend to the sequence in reverse order, accurately predicting the parity of the majority of the sequence simultaneously during the final internal ticks. This reverse search through the data suggests that the CTM is carrying out a form of planning, building up its understanding of the observed data before making a final decision on the cumulative parity of the sequence. These results highlight that although multiple strategies exist for solving this task, some of which are more interpretable than others, the CTM clearly demonstrates the ability to form and follow a strategy.

8.2. Demonstrations

We show two demonstrations in Figure 21. The first example (top) depicts a typical sample from the dataset, which contains values of 1 and −1 at random positions. In this case, the CTM perfectly predicts the cumulative parity. The dynamics of the attention heads (a) shows that the attention moves sequentially through the input data, in agreement with Figure 20. Furthermore, we can see that some heads attend to only positive or negative values, while other heads attend to both. The second example (bottom) shows a failure case of the model. When presented with an input sequence containing only positive parities, the model struggles to accurately predict the cumulative parity, highlighting an edge-case limitation.



(a) Attention trajectories (b) Prediction (c) Target

(d) Attention trajectories (e) Prediction (f) Target

Figure 21 | Visualization of attention dynamics for two sample solutions to the parity task. The top row (a, b, c) depicts a perfect prediction, while the bottom row depicts a failure case (d, e, f). Attention trajectories (i.e., positions of the argmax in the attention weights) for each of the eight attention heads are shown in (a, d). Each point indicates the input position with the highest attention weight at a given timestep. Color represents progression over time, with brighter colors indicating later internal ticks. The cross (×) marks the timestep at which the model reached maximum certainty in its prediction. Attention heads typically move sequentially through input positions. Certain heads consistently attend only to positive or negative input values, whereas others alternate between positive and negative input values. Similarly, some heads remain relatively static, while others move quickly over the data. (b, e) The model's predictions. (c, f) The target.

9. Q&A MNIST

To assess the CTM's capabilities for memory, retrieval, and arithmetic computation, we devise a Question and Answering (Q&A) MNIST task, reminiscent of Manhaeve et al. (2018) or Schlag and Schmidhuber (2021). In this task, the model sequentially observes a series of MNIST digits (LeCun et al., 1998), followed by an interwoven series of index and operator embeddings that determine which of the observed digits to select and which modular operation to perform over them. This allows us to probe whether the CTM can simultaneously recognize hand-drawn digits, recall previous observations, and perform logical computation on them without any prior knowledge of the digits depicted in the images or the relationships between them. Furthermore, by applying more operations at inference time than observed during training time, we can test the generalizability of the CTM.

Specifically, the model first observes N_d MNIST digits sequentially for t_d internal ticks each. Next, the model receives an interwoven sequence of N_{idx} index embeddings (indicating which digit to select) and N_{op} operator embeddings (specifying either modular addition or subtraction, where each intermediate result is taken modulo 10 to keep answers within the range 0–9), each presented for t_{idx} and t_{op} internal ticks, respectively. Finally, the model observes a zero tensor for t_{ans} internal ticks, signaling the model to produce its answer. The target, between 0 and 9, results from the composition of all specified modular arithmetic operations. An example is shown in Figure 22.

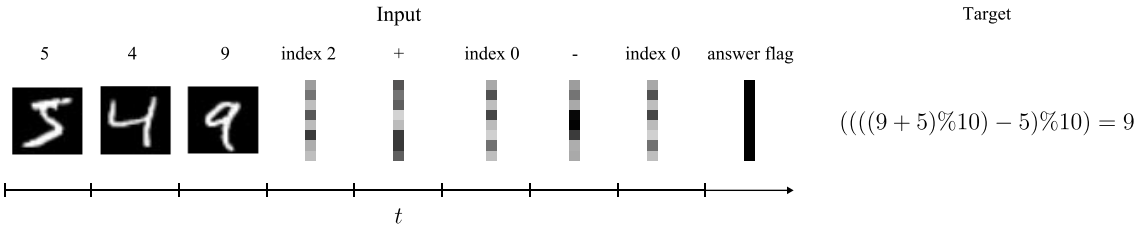


Figure 22 | Overview of the Q&A MNIST task. The model observes a series of digits followed by a series of index and operator embeddings, each repeated for several internal ticks. The model is then shown an answer flag and must predict the result of the modular operations.

We trained CTMs and parameter-matched LSTMs with two different configurations, varying how many internal ticks were used to process each input. Digits and embeddings were observed for either 1 or 10 internal ticks, with corresponding answering times of 1 or 10 internal ticks. The number of digits and the number of operations were sampled uniformly between 1 and 4. Memory lengths for the 1 and 10 internal ticks per input CTMs were set to 3 and 30 steps, respectively. We highlight that with these observation and memory length configurations that the digit observations will always lie outside of the memory length-sized window during the answering stage. In this way, the CTM must organize its activations such that it can recall the digits at later time steps. The CTM is trained with the loss defined in Section 2.5, computed only over the final t_{ans} steps. Once more, we set t_2 to be the final iteration for the LSTM for stable training. A full overview can be found in Appendix H.

9.1. Results

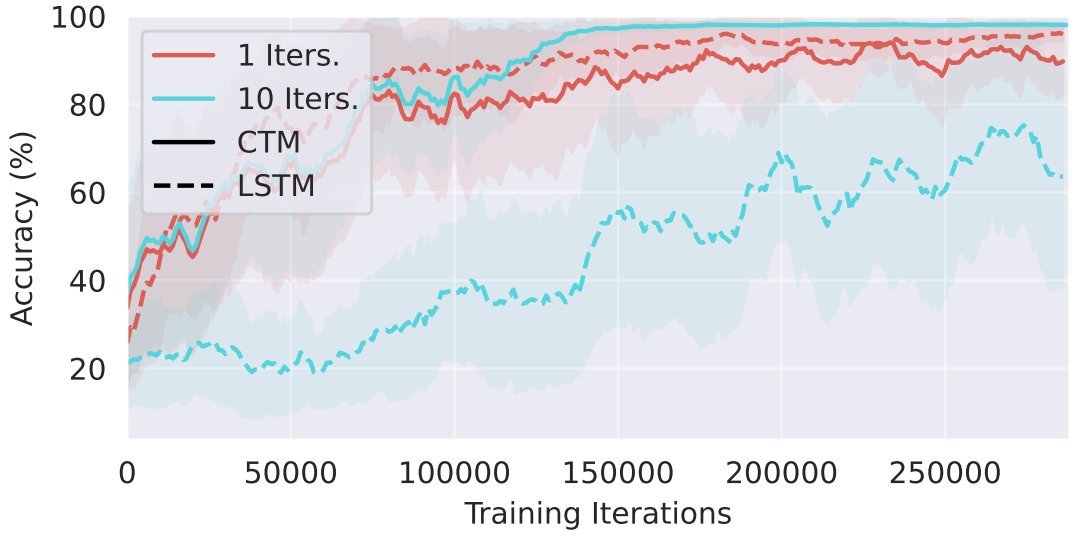
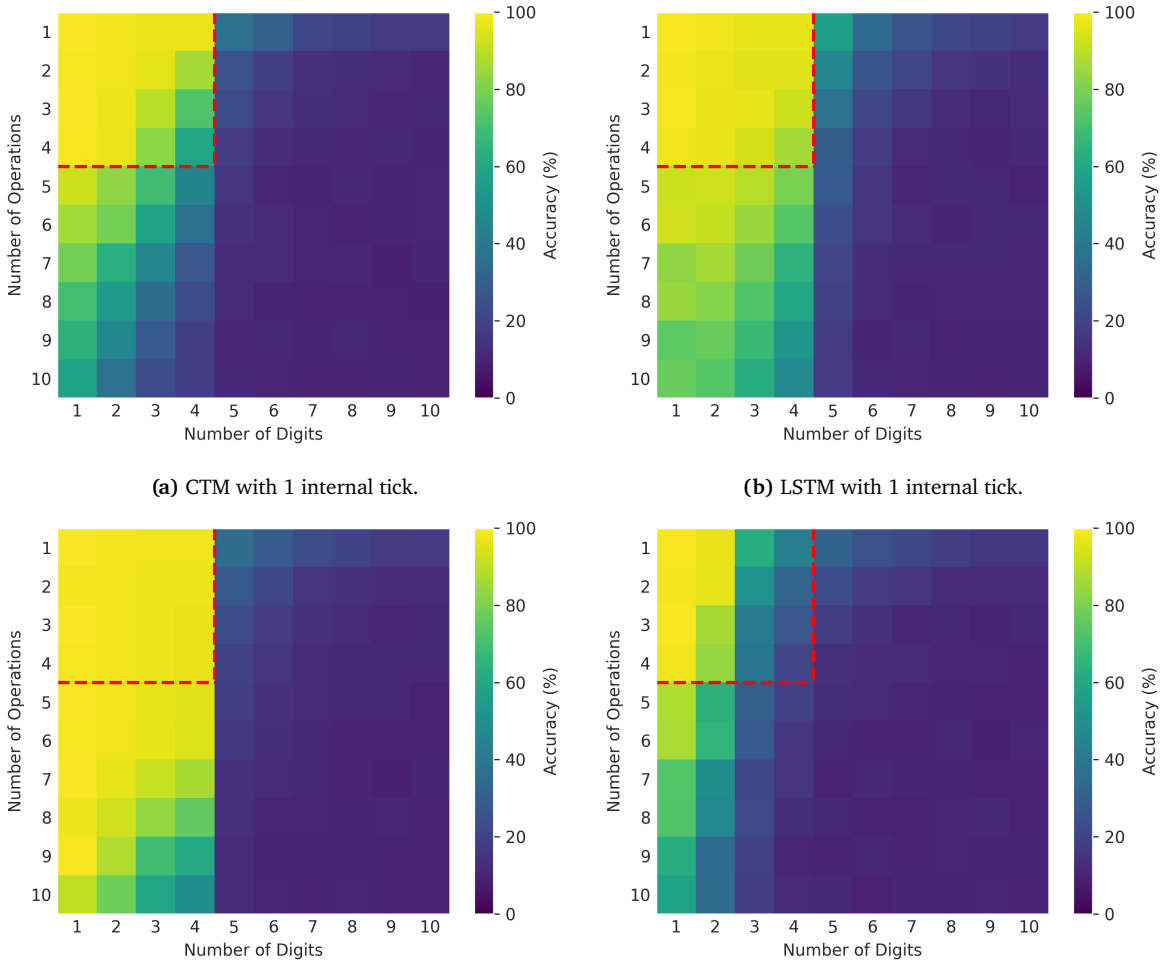


Figure 23 | Training curves for the CTM and LSTM on the Q&A MNIST task. The shaded areas represent one standard deviation across seeds. With a single internal tick, the LSTM outperforms the CTM. However, the performance of the CTM increases with the number of internal ticks, while the LSTM becomes increasingly unstable.

Memory via synchronization. Training curves for three seeded runs for CTMs and parameter-matched LSTMs are shown in Figure 23. With a single internal tick, the LSTM initially outperforms the CTM. As the number of internal ticks increases, the LSTM's performance degrades and learning becomes considerably more unstable. In contrast, the CTM consistently improves its performance with additional thinking time. Specifically, all three seeded runs for the CTM with 10 internal ticks per input achieved over 96% accuracy on the most challenging in-distribution task (performing four operations after observing four digits). In contrast, the corresponding 10-internal tick LSTM performed at or below 21% accuracy across all seeded runs. The strong performance of the single-tick LSTMs highlights the effectiveness of the LSTM's complex gated update, however, this mechanism does not scale effectively to multiple internal steps, unlike the CTM, which effectively utilizes internal ticks to build up a synchronization representation.

The CTM performs well even when the observed digits are outside of the memory window, indicating that it has learned to memorize what it has observed to some degree, purely via the organization and synchronization of neurons. The strong performance of the CTM indicates that processing timing information through the synchronization of neuron activations may be a powerful mechanism for memorization and recall.



(a) CTM with 1 internal tick.

(b) LSTM with 1 internal tick.

(c) CTM with 10 internal ticks.

(d) LSTM with 10 internal ticks.

Figure 24 | Generalizability on the Q&A MNIST task for CTM and LSTM models with 1 and 10 internal ticks. The x-axis denotes the number of MNIST digits input to the model, the y-axis denotes the number of operations the model must perform, and the color corresponds to the test accuracy.

The CTM can generalize. We examine generalization by measuring the accuracy of the models when given more digits or index-operator embeddings than used during training. Figure 24 shows the accuracy of the CTM and LSTM as a function of the number of digits shown and operations to perform, with the training regime highlighted in red. We find that both the CTM and LSTM baselines can generalize to an increased number of operations. To understand how the model is capable of generalizing out of distribution, we illustrate an example thought process of the CTM in Figure 25, which shows a sample sequence of inputs and a snapshot of the output logits. We find that the CTM sequentially computes the modular computation as the embeddings are observed, instead of waiting for the final answer flag to determine the final solution at once. A similar behavior can be seen in the 1-internal tick LSTM baseline. We are not claiming that the CTM can do something that the LSTM cannot, but instead that it can learn to **use synchronization as a tool** to solve this task, and that the result is both effective and scales to longer task requirements.

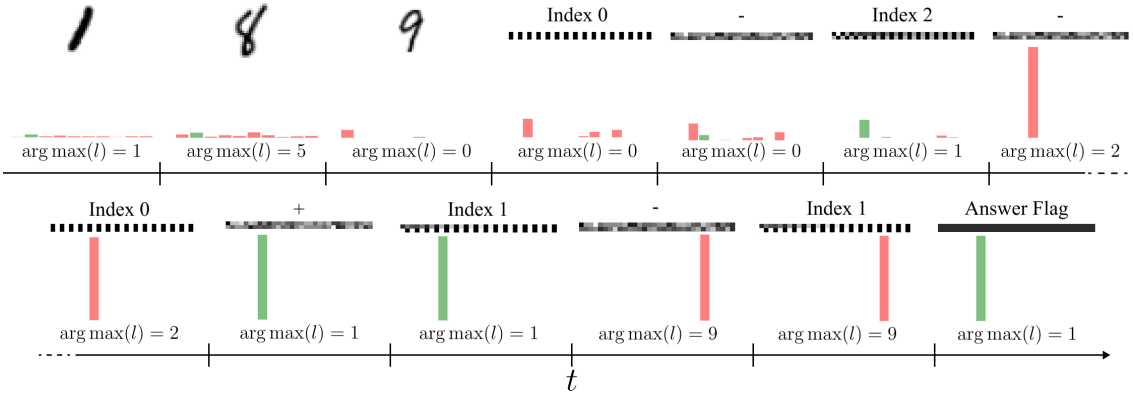
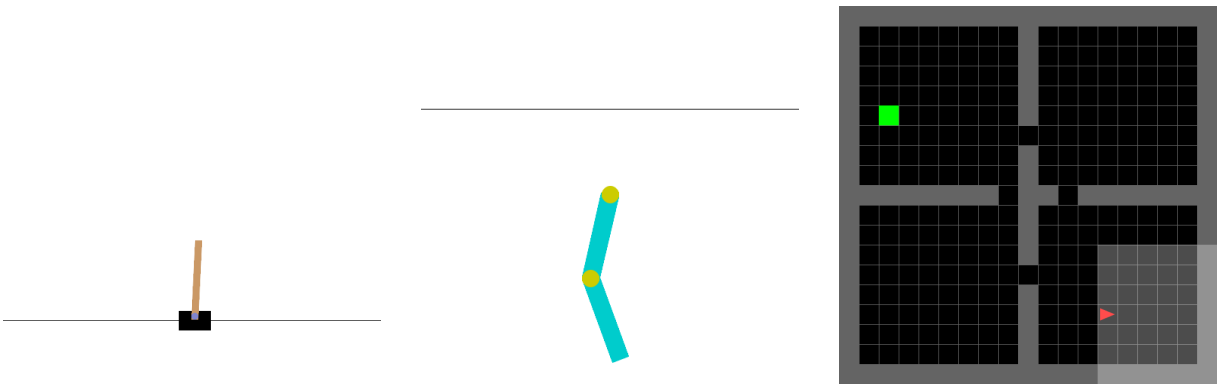


Figure 25 | Example CTM thought process from the Q&A MNIST task. Shown are the inputs to the model (MNIST digit, index and operator embeddings) as well as the argmax of the output logits l, at different snapshots. Each input is repeated for 10 internal ticks. In this case, the model is to compute ((((((1 − 9)%10) − 1)%10 + 8)%10 − 8)%10). We find that the model computes each part of this composition sequentially as the embeddings are observed, with outputs of 2, 1, 9, and finally, the correct answer of 1, projected from the synchronization representation.

10. Reinforcement learning



(a) CartPole **(b)** Acrobot **(c)** MiniGrid Four Rooms

Figure 26 | Reinforcement learning environments. In CartPole (a), the agent balances a pole on a cart using two discrete actions (left or right). Observations include two non-masked inputs: cart position and pole angle. In Acrobot (b), the agent applies one of three torques (+1, −1, or 0) to a two-joint arm to raise its end above a target height, using four non-masked inputs (sines and cosines of joint angles). In MiniGrid Four Rooms (c), the agent navigates using seven discrete actions (e.g., turn left, turn right, etc.) and observes a $3 \times 7 \times 7$ input tensor encoding object, color, and state IDs within a 7×7 limited field of view.

We have previously shown that the CTM can process sequentially on non-sequential tasks via its decoupled internal recurrence. Here, we extend the CTM to sequential decision-making tasks involving interactions with external environments. Specifically, we train CTMs using reinforcement learning (RL), where the model learns action-selection policies based on environmental observations and trial-and-error interactions. In this setting, the CTM processes one or more internal ticks before producing an action that transitions the environment to the next state. To achieve this, we continuously maintain the neuron dynamics across these internal ticks over successive environment steps, allowing previous environmental observations to influence current internal states via the NLMs. A central goal of this section is to provide evidence that the CTM can be set up to learn in a continuous environment.

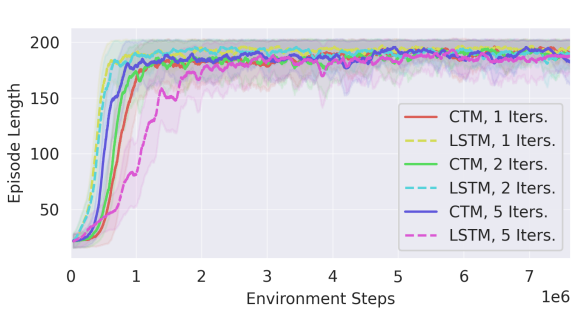
Environments. We test the CTM on two classic control tasks and one navigation task, namely, CartPole, Acrobot and MiniGrid Four Rooms, implemented in Gymnasium (Barto et al., 1983; Chevalier-Boisvert et al., 2023; Sutton, 1995; Towers et al., 2024). Examples of these tasks are shown in Figure 26. Because the CTM maintains an activation history across environment transitions, it functions as a stateful recurrent neural network. Therefore, we specifically evaluate the CTM in partially observable settings, where RNNs are effective (Hausknecht and Stone, 2015). Partial observability is introduced by masking the positional and angular velocity observation components in the control tasks and restricting the field of view in the navigation task. This masking converts these tasks into partially observable Markov decision processes (POMDPs), requiring the CTMs to develop policies that recall past observations. For instance, in the Acrobot task, selecting the correct action depends on recalling past positions and inferring the velocity to increase arm elevation.

Architecture. For the RL tasks, the following architecture is used and trained with Proximal Policy Optimization (Schulman et al., 2017). First, the input observations are processed using a series of fully connected layers. For the navigation task, this also includes embedding the observed states and adding positional embeddings, corresponding to locations within the agent's field of view. This representation is then processed by the CTM, without the use of an attention mechanism, for a number of internal ticks, before the synchronization vector is output and processed by the actor and critic heads. We compare this approach with parameter-matched LSTMs baselines, where the internal ticks are instead processed by an LSTM cell, which outputs its hidden state to the actor and critic networks. The purpose of such comparison is not too showcase a superiority of one architecture over another, but to show that a CTM can leverage the synchronization of a continuous history of activations, and achieve a comparable performance to LSTMs. A full description of the architecture and optimization hyperparameters can be found in Appendix I.

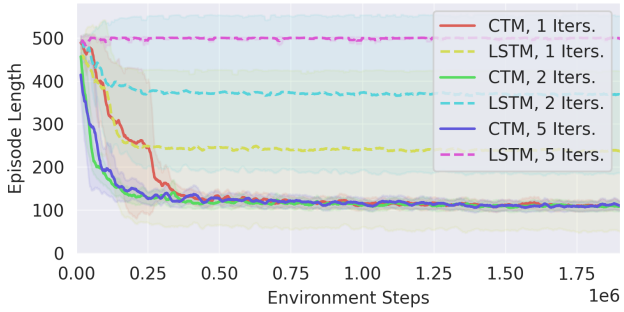
10.1. Results

The CTM can continuously interact with the world. Training curves for the reinforcement learning tasks are shown in Figure 27. In all tasks, we find that the CTM achieves a similar performance to the LSTM baselines.

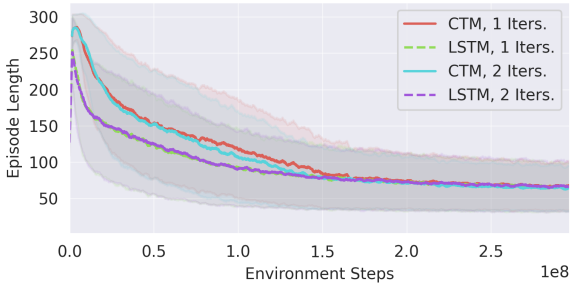
Figure 28 compares the neuron traces of the CTM and the LSTM baselines for the CartPole, Acrobot and MiniGrid Four Rooms tasks. In the classic control tasks, the activations for both the CTM and the LSTM feature oscillatory behavior, corresponding to the back-and-forth movements of the cart and arm. For the navigation task, a rich and complex activation pattern emerges in the CTM. The LSTM on the other hand, features a less diverse set of activations. The LSTMs trained in this section have a more dynamic neural activity than what can be seen when trained on CIFAR-10 (Figure 12). This is likely due to the sequential nature of RL tasks, where the input to the model changes over time owing to its interaction with the environment, inducing a feedback loop that results in the model's latent representation also evolving over time.



(a) Cartpole

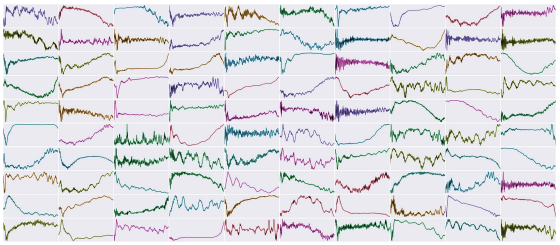


(b) Acrobot

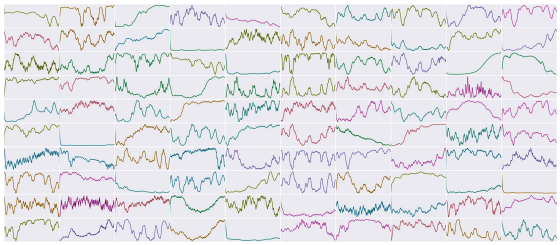


(c) MiniGrid 4-Rooms

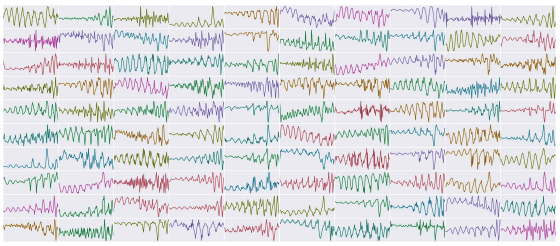
Figure 27 | Training curves for reinforcement learning tasks. Each curve depicts a moving average of the episode length during training, averaged over three training runs. The shaded region represents one standard deviation across seeds. For Cartpole, higher is better. For Acrobot and MiniGrid 4-rooms, lower is better.



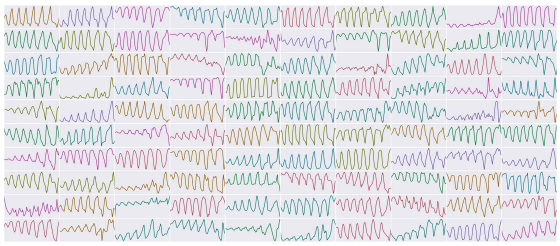
(a) CTM, CartPole, 2 Internal Ticks



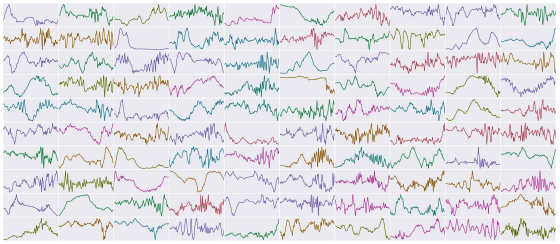
(b) LSTM, CartPole, 2 Internal Ticks



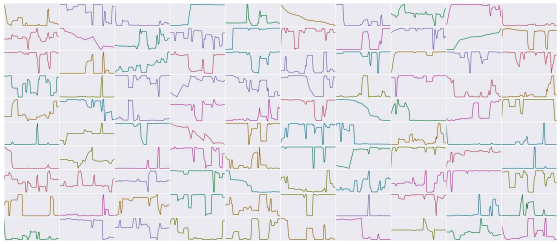
(c) CTM, Acrobot, 1 Internal Tick



(d) LSTM, Acrobot, 1 Internal Tick



(e) CTM, MiniGrid Four Rooms, 2 Internal Ticks



(f) LSTM, MiniGrid Four Rooms, 2 Internal Ticks

Figure 28 | Neural activities over the course of a single episode for the CTM and LSTM on CartPole, Acrobot and MiniGrid Four Rooms tasks. The CTM features richer neuron dynamics than the LSTM.

11. Related work

Modern neural networks have achieved remarkable success across a range of domains, yet they typically rely on fixed-depth, feedforward computation, with limited flexibility to adapt their processing based on input complexity. In contrast, biological brains exhibit dynamic neural activity that unfolds over time, adjusting computation to the demands of a task. The CTM builds on this idea by explicitly modeling internal neural timing and synchronization. In this section, we highlight key works related to *adaptive computation*, *iterative reasoning*, and *biologically inspired architectures*, all of which inform the motivation behind the CTM.

11.1. Adaptive computation and dynamic halting

A number of approaches have explored *adaptive computation*, where the number of inference steps varies based on input difficulty or confidence. Early-exit networks (e.g., Bolukbasi et al. (2017)) allow models to terminate inference early if intermediate layers produce confident predictions, thereby saving computation on easy examples. PonderNet (Banino et al., 2021) introduced a stochastic halting mechanism for recurrent models, enabling learned per-input "ponder times" through an end-to-end differentiable loss that balances accuracy and efficiency. This method improved upon Adaptive Computation Time (ACT) (Graves, 2016) by offering more stable training and stronger generalization on algorithmic reasoning tasks.

More recently, AdaTape (Xue et al., 2023) proposed a flexible memory-augmented architecture that dynamically extends the input with additional "tape tokens," effectively granting the model more computation budget as needed. Similarly, the Sparse Universal Transformer (SUT) (Tan et al., 2023) combines recurrent weight sharing with dynamic halting and Mixture-of-Experts routing, enabling the model to apply varying numbers of recurrent transformer layers per input. These methods demonstrate the benefits of input-dependent computation, aligning compute cost with problem difficulty—a goal also pursued by the CTM via its internal "thought" dimension.

11.2. Iterative and recurrent reasoning

The CTM shares common ground with models designed for *iterative reasoning* and *internal recurrence*. Quiet-STaR (Zelikman et al., 2024), for example, teaches language models to "think before speaking" by inserting hidden rationale tokens during training, encouraging internal computation before output generation. This process enhances performance on complex tasks like math reasoning and commonsense QA. Other architectures like Recurrent Independent Mechanisms (RIMs) (Goyal et al., 2019) split computation across sparsely activated, modular sub-networks, which evolve asynchronously over time, improving systematic generalization and multi-step reasoning. These approaches echo the CTM's aim of simulating thought through internal, decoupled computation that is not directly tied to the input sequence.

11.3. Biologically inspired neural dynamics

A growing body of work aims to make neural computation more biologically plausible (Schmidgall et al., 2024). Liquid Time-Constant Networks (LTCNs) (Hasani et al., 2021) use neurons governed by time-varying differential equations, enabling each neuron to adapt its response dynamically based on input history. These networks have shown strong performance on temporal tasks with high sample efficiency and robustness. Similarly, Spiking Neural Networks (SNNs) have gained traction as biologically grounded alternatives to standard deep networks, relying on discrete spikes and precise timing to encode and process information. Recent advances (e.g., Stan and Rhodes (2024)) combine SNNs with state-space models and synchronization mechanisms, achieving competitive or even superior performance on long-range sequence tasks.

The CTM draws inspiration from such neural mechanisms – particularly *temporal coding* and *neural*

synchrony – to encode information in the timing of activity rather than static activations. Unlike prior models, however, the CTM uses these dynamics directly as a latent representation for attention and output generation, leading to a unified architecture where reasoning emerges naturally from the interplay of evolving neuron states.

12. Discussion and future work

In this technical report we presented the CTM as a model that unfolds and leverages the temporal dynamics of neural activity as the primary mechanism for its intelligence. To our best knowledge, using **neural synchronization over time as the latent representation** for a model has never been accomplished, particularly at this scale. The CTM is an instantiation of a model that uses the precise interplay and timing of temporal dynamics – which is believed to be crucial in natural cognition – to successfully perform a diverse range of tasks, for which we gave evidence herein.

Our goal for this work was to introduce this new model, and the perspective that **neural dynamics can be a powerful tool for neural computation**. We hope that the research community can adopt aspects of our work to build more biologically plausible and performant AI. In the following subsections we offer some discussion and perspectives based on our observations.

12.1. Intuitive perspective, biological plausibility

The CTM's outputs, $\mathbf{y}^t$, are computed as linear projections from synchronization. Consider, for example, object classification, where $\mathbf{y}^t$ represents class predictions. In this scenario, the CTM must observe abstract features in the input data to produce specific **activation dynamics** within its neurons. These activation dynamics, in turn, must synchronize in a precise manner to generate accurate predictions. Intuitively, this implies that the CTM learns to develop **persistent patterns of neural activity** over internal ticks in response to the input data, effectively building up its output through a temporal process. This concept aligns with recent notions of reasoning and is a key motivation behind our choice of the term 'thought.' Furthermore, such a dynamic and temporal representation contrasts sharply with existing methods that use standard representations. While the experiments in this paper represent an initial exploration of the potential of this type of representation, its closer resemblance to biological processes suggests that its eventual utility could be substantial.

12.2. The strengths of synchronization over time

Multi-resolution. Our measurement of synchronization depends on activity over time, but not exactly on time itself. This potentially frees up some portion of synchronization to change slowly over time, while our mechanism for learnable decaying time dependency (see Section 2.4) enables the emergence of short-term dependence. The result is that synchronization is a representation that can capture events or perspectives at any number of resolutions. Many real-world scenarios present themselves as situations where features or ideas where such a multi-resolution perspective could be powerful; we will explore this in future work.

Memory Further to this point is that a given synchronization captures not only the CTM's cognition over some time period, but also the consequence of it taking action, owing to the recurrent cycle of internal ticks. Such a perspective is far more akin to an 'experience' than what a snapshot representation could yield. Hence, synchronization matrices as memories presents an interesting avenue for future work.

Plasticity and gradient-free learning. As we have defined it in this technical report, synchronization measures how neurons fire together, which is a very Hebbian-like perspective (Hebb, 2005; Najarro

and Risi, 2020). Leveraging this notion to explore lifelong learning (Kudithipudi et al., 2022; Wang et al., 2024), plasticity, or even gradient-free optimization are exciting avenues for future work.

Cardinality The full synchronization in a D dimensional CTM is $(D \times (D+1))/2$ dimensional (the upper triangle of the full synchronization matrix). Research suggests that large representations are advantageous for a number of reasons (Allen-Zhu et al., 2019; Frankle and Carbin, 2018). Synchronization gives us access to a large and meaningful representation space at no additional cost. We intend to explore what utility can be gained from such a high-cardinality representation space, particularly in the realm of multi-modal modeling.

12.3. Continuous worlds

We took inspiration from nature to build the CTM, but used well-established protocols and datasets when training it. These datasets and protocols, however, are not necessarily natural. For instance, independent and identically distributed data is expected when training conventional NNs, but that is not how the real-world operates. Events happen over time and are usually arranged accordingly. Hence, we wish to move toward training the CTM in a biologically plausible fashion for future work. Application to sequential data (e.g., videos, text), particularly when sampled in order during training, is a promising avenue for future work.

Language modeling. We have yet to apply the CTM to the task of language modeling, but it is straightforward to adapt it to ingesting and thinking about text given that it uses attention. Moreover, since it can build a world model and navigate it (see Section 4), the CTM could potentially do without positional encodings, instead building a contextualized 'world model' of what it observes. One potential avenue we encourage readers to explore is that of applying the CTM to pretrained language models. For future work, we will build and train custom CTMs on text data in order to understand its capability in that domain.

12.4. What failed, and how did we get here?

We believe it is important to explain what we initially tried as the core representation for the CTM (as opposed to synchronization). The activated latent space, $\mathbf{z}^t$, is an obvious candidate. We found, however, that the dynamic and complex nature of $\mathbf{Z}$ meant we needed to perform some smoothing operations (e.g., accumulating some 'holder' latent or logits). Further, given that this representation is strongly coupled to t, we found that the CTM would learn when exactly to produce an output (i.e., when the loss function was applied) instead of relying on the internal self-organization as the primary driving force.

By adding neuron timing back into the system we introduced this challenge; thankfully, synchronization, being time-independent, proved an elegant solution to overcoming these challenges. The learnable decaying time-dependence also offers a neat solution whereby the CTM can learn to interact with its world based on short-term neural behavior, should this be preferred.

13. Conclusion

The Continuous Thought Machine (CTM) represents a novel step towards bridging computational efficiency with biological plausibility in artificial intelligence. By moving beyond traditional pointwise activation functions to private neuron-level models, the CTM cultivates far richer neuron dynamics. Crucially, it leverages neural synchronization as a powerful and fundamentally new type of representation – distinct from the activation vectors prevalent since the early days of neural networks. This direct use of neuron dynamics as a first-class representational citizen allows the CTM to exhibit behaviors qualitatively different from contemporary models.

Our research demonstrates the tangible benefits of this approach. The CTM can dynamically build representations over time for tasks like image classification, form rich internal maps to attend to specific input data without positional embeddings, and naturally exhibit adaptive computation. Furthermore, it learns to synchronize neural dynamics to store and retrieve memories beyond its immediate activation history. This internal processing also lends itself to greater interpretability, as seen in its methodical solving of mazes and parity tasks.

Remarkably, the core CTM architecture remained largely consistent across a diverse range of challenging tasks, requiring only input/output module adjustments. This versatility and trainability were particularly evident in complex scenarios like maze navigation. The CTM succeeded with minimal tuning, where a traditional model like the LSTMs still struggled even after significant tuning efforts.

This work underscores a vital, yet often underexplored, synergy between neuroscience and machine learning. While modern AI is ostensibly brain-inspired, the two fields often operate in surprising isolation. The CTM serves as a testament to the power of drawing inspiration from biological principles. By starting with such inspiration and iteratively following the emergent, interesting behaviors, we developed a model with unexpected capabilities, such as its surprisingly strong calibration in classification tasks, a feature that was not explicitly designed for.

It is crucial to note that our approach advocates for borrowing concepts from biology rather than insisting on strict, literal plausibility; real neurons may not access their activation history as modeled in the CTM, yet emergent phenomena like traveling waves still manifest. This nuanced balance between practicality and biological inspiration opens a landscape of new research directions, which may hold the key to unlocking capabilities currently missing in AI, potentially leading to systems that exhibit more human-like intelligence and address its current limitations.

When we initially asked, “why do this research?”, we hoped the journey of the CTM would provide compelling answers. By embracing light biological inspiration and pursuing the novel behaviors observed, we have arrived at a model with emergent capabilities that exceeded our initial designs. We are committed to continuing this exploration, borrowing further concepts to discover what new and exciting behaviors will emerge, pushing the boundaries of what AI can achieve.

Limitations

The main limitation of the CTM is that it requires sequential processing that cannot be parallelized, meaning that training times are longer than a standard feed-forward model, particularly when considering that the current state of modern AI models are well suited to the hardware and software that has developed in parallel (Hooker, 2021).

The additional parameter cost associated with neuron-level models can also be considered a limitation. Whether the benefits outweigh the costs remains to be proven, but we believe their utility can be high.

References

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. In *International conference on machine learning*, pages 242–252. PMLR, 2019.

Cristina M Atance and Daniela K O'Neill. Episodic future thinking. *Trends in cognitive sciences*, 5(12): 533–539, 2001.

Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021.

Arpit Bansal, Avi Schwarzschild, Eitan Borgnia, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. End-to-end algorithm synthesis with recurrent networks: Extrapolation without overthinking. *Advances in Neural Information Processing Systems*, 35:20232–20242, 2022.

Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077.

Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning*, pages 527–536. PMLR, 2017.

Natalia Caporale and Yang Dan. Spike timing–dependent plasticity: a hebbian learning rule. *Annu. Rev. Neurosci.*, 31(1):25–46, 2008.

Peter Cariani and Janet M Baker. Time is of the essence: neural codes, synchronies, oscillations, architectures. *Frontiers in Computational Neuroscience*, 16:898829, 2022.

Makram Chahine, Ramin Hasani, Patrick Kao, Aaron Ray, Ryan Shubert, Mathias Lechner, Alexander Amini, and Daniela Rus. Robust flight navigation out of distribution with liquid neural networks. *Science Robotics*, 8(77):eadc8892, 2023.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. Arc prize 2024: Technical report. *arXiv preprint arXiv:2412.04604*, 2024.

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International conference on machine learning*, pages 933–941. PMLR, 2017.

Rahul Dey and Fathi M Salem. Gate-variants of gated recurrent unit (gru) neural networks. In *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*, pages 1597–1600. IEEE, 2017.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *arXiv preprint arXiv:2502.05171*, 2025.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. MIT press Cambridge, 2016.

James Gornet and Matt Thomson. Automated construction of cognitive maps with visual predictive coding. *Nature Machine Intelligence*, 6(7):820–833, 2024.

Anirudh Goyal, Alex Lamb, Jordan Hoffmann, Shagun Sodhani, Sergey Levine, Yoshua Bengio, and Bernhard Schölkopf. Recurrent independent mechanisms. *arXiv preprint arXiv:1909.10893*, 2019.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International conference on machine learning*, pages 1321–1330. PMLR, 2017.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

Ramin Hasani, Mathias Lechner, Alexander Amini, Daniela Rus, and Radu Grosu. Liquid time-constant networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 7657–7666, 2021.

Matthew J Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *AAAI fall symposia*, volume 45, page 141, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

Tien Ho-Phuoc. Cifar10 to compare visual recognition performance between deep neural networks and humans. *arXiv preprint arXiv:1811.07270*, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Patrick Hohenecker and Thomas Lukasiewicz. Ontology reasoning with deep neural networks. *Journal of Artificial Intelligence Research*, 68:503–540, 2020.

Sara Hooker. The hardware lottery. *Communications of the ACM*, 64(12):58–65, 2021.

Michael Igorevich Ivanitskiy, Rusheb Shah, Alex F. Spies, Tilman Räuker, Dan Valentine, Can Rager, Lucia Quirke, Chris Mathwin, Guillaume Corlouer, Cecilia Diniz Behn, and Samy Wu Fung. A configurable library for generating and manipulating maze datasets, 2023. URL http://arxiv.org/abs/2309.10498.

Mozes Jacobs, Roberto C Budzinski, Lyle Muller, Demba Ba, and T Anderson Keller. Traveling waves integrate spatial information into spectral representations. *arXiv preprint arXiv:2502.06034*, 2025.

Herbert Jaeger. Echo state network. *scholarpedia*, 2(9):2330, 2007.

Andrew Jaegle, Felix Gimeno, Andy Brock, Oriol Vinyals, Andrew Zisserman, and Joao Carreira. Perceiver: General perception with iterative attention. In *International conference on machine learning*, pages 4651–4664. PMLR, 2021.

Louis Kirsch and Jürgen Schmidhuber. Meta learning backpropagation and improving it. *Advances in Neural Information Processing Systems*, 34:14122–14134, 2021.

Louis Kirsch, Sebastian Flennerhag, Hado Van Hasselt, Abram Friesen, Junhyuk Oh, and Yutian Chen. Introducing symmetries to black box meta reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 7202–7210, 2022.

Dhireesha Kudithipudi, Mario Aguilar-Simon, Jonathan Babb, Maxim Bazhenov, Douglas Blackiston, Josh Bongard, Andrew P Brna, Suraj Chakravarthi Raja, Nick Cheney, Jeff Clune, et al. Biological underpinnings for lifelong learning machines. *Nature Machine Intelligence*, 4(3):196–210, 2022.

Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40:e253, 2017.

Yann LeCun. A path towards autonomous machine intelligence version 0.9. 2, 2022-06-27. *Open Review*, 62(1):1–62, 2022.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Wolfgang Maass. On the relevance of time in neural computation and learning. *Theoretical Computer Science*, 261(1):157–178, 2001.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

Gary Marcus. Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631*, 2018.

Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.

Takeru Miyato, Sindy Löwe, Andreas Geiger, and Max Welling. Artificial kuramoto oscillatory neurons. *arXiv preprint arXiv:2410.13821*, 2024.

Lyle Muller, Frédéric Chavane, John Reynolds, and Terrence J Sejnowski. Cortical travelling waves: mechanisms and computational principles. *Nature Reviews Neuroscience*, 19(5):255–268, 2018.

Elias Najarro and Sebastian Risi. Meta-learning through hebbian plasticity in random networks. *Advances in Neural Information Processing Systems*, 33:20719–20731, 2020.

Joachim Pedersen, Erwan Plantec, Eleni Nisioti, Milton Montero, and Sebastian Risi. Structurally flexible neural networks: Evolving the building blocks for general agents. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1119–1127, 2024.

Joshua C Peterson, Ruairidh M Battleday, Thomas L Griffiths, and Olga Russakovsky. Human uncertainty makes classification more robust. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9617–9626, 2019.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, et al. Humanity's last exam, 2025. URL https://arxiv.org/abs/2501.14249.

Jing Ren and Feng Xia. Brain-inspired artificial intelligence: A comprehensive review. *arXiv preprint arXiv:2408.14811*, 2024.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015.

Imanol Schlag and Jürgen Schmidhuber. Augmenting classic algorithms with neural components for strong generalisation on ambiguous and high-dimensional data. In *Advances in Programming Languages and Neurosymbolic Systems Workshop*, 2021.

Samuel Schmidgall, Rojin Ziaei, Jascha Achterberg, Louis Kirsch, S Hajiseyedrazi, and Jason Eshraghian. Brain-inspired learning in artificial neural networks: a review. *APL Machine Learning*, 2(2), 2024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Furong Huang, Uzi Vishkin, Micah Goldblum, and Tom Goldstein. Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. *Advances in Neural Information Processing Systems*, 34:6695–6706, 2021.

Matei-Ioan Stan and Oliver Rhodes. Learning long sequences in spiking neural networks. *Scientific Reports*, 14(1):21957, 2024.

Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems*, 8, 1995.

Shawn Tan, Yikang Shen, Zhenfang Chen, Aaron Courville, and Chuang Gan. Sparse universal transformer. *arXiv preprint arXiv:2310.07096*, 2023.

Neil C Thompson, Kristjan Greenewald, Keeheon Lee, Gabriel F Manso, et al. The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*, 10, 2020.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024. URL https://arxiv.org/abs/2407.17032.

Peter Uhlhaas, Gordon Pipa, Bruss Lima, Lucia Melloni, Sergio Neuenschwander, Danko Nikolić, and Wolf Singer. Neural synchrony in cortical networks: history, concept and current status. *Frontiers in integrative neuroscience*, 3:543, 2009.

Paul E Utgoff. *Machine learning of inductive bias*, volume 15. Springer Science & Business Media, 2012.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Vasilis Vryniotis and Matthieu Cord. How to Train State-Of-The-Art Models Using TorchVision's Latest Primitives. PyTorch Blog, dec 2021. URL https://pytorch.org/blog/how-to-train-state-of-the-art-models-using-torchvision-latest-primitives/. Last edited on 2021-12-22.

Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057. PMLR, 2015.

Fuzhao Xue, Valerii Likhosherstov, Anurag Arnab, Neil Houlsby, Mostafa Dehghani, and Yang You. Adaptive computation with elastic input sequence. In *International Conference on Machine Learning*, pages 38971–38988. PMLR, 2023.

Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. *arXiv preprint arXiv:2311.12424*, 2023.

Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D Goodman. Quiet-star: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*, 2024.

Tao Zhang, Jia-Shu Pan, Ruiqi Feng, and Tailin Wu. T-scend: Test-time scalable mcts-enhanced diffusion model. *arXiv preprint arXiv:2502.01989*, 2025.

A. Glossary

Term	Description
Internal tick	One step of internal computation.
Memory length	Length of rolling history of pre-activations, updated in a rolling FIFO fashion
Synapse model	Recurrent model that takes $\mathbf{z}^t$ and $\mathbf{o}^t$ as input to preduct pre-activations, $\mathbf{a}^t$.
Pre-activations	Output of recurrent synapse model, input to NLMs.
Post-activations	Output of NLMs, neuron states at time t.
(Pre/Post) activation history	Sequentially ordered history of activations over time.
Neuron-Level Model (NLM)	Per-neuron MLP over pre-activation history.
Synchronization	Dot product of post-activation histories.
Self-pair	Diagonal synchronization matrix entries (i, i).
Action synchronization	Synchronization representation for attention queries.
Output synchronization	Synchronization representation for predictions.
Decay r_{ij}	Learnable time decay for synchronization (action or output).
Feature extractor	Task-specific input encoder (e.g., ResNet).
Attention output	Output after cross-attention using queries, $\mathbf{q}^t$, computed from action synchronization, and keys/values from data.

Table 1 | Glossary of terms.

Symbol	Meaning
T	Number of internal ticks
M	Memory length
d_{model}	Dimensionality of latent state in the CTM
d_{input}	Dimensionality of attention output
d_{hidden}	Hidden size in each neuron's private MLP (NLM)
k	Depth of the synapse MLP or U-Net
p_{dropout}	Dropout probability in the synapse model
n_{heads}	Number of heads in multi-head attention
J_{action}	Number of neurons used for action synchronization
J_{out}	Number of neurons used for output synchronization
D_{action}	Dimensionality of action synchronization vector
D_{out}	Dimensionality of output synchronization vector
n_{self}	Number of self-pairs (i, i) used in synchronization sampling
r_{ij}	Learnable decay parameter for synchronization between neuron i and j
$\mathbf{S}^t$	Full synchronization matrix at internal tick t
$\mathbf{q}^t$	Query vector projected from action synchronization
$\mathbf{y}^t$	Output vector (e.g., logits) projected from output synchronization

Table 2 | Glossary of symbols.

B. Method details

B.1. Synapse models

Figure 29 show the synapse model which is the recurrent structure that shares information across neurons in the CTM. It is implemented by choosing a depth of k (always even), where each subsequent

layer width is chosen to linearly reduce the dimensionality until a width of 16 is reached, and then increase thereafter, using skip connections to retain information. The synapse model also takes $\mathbf{o}^t$ (the output of attention) as input.

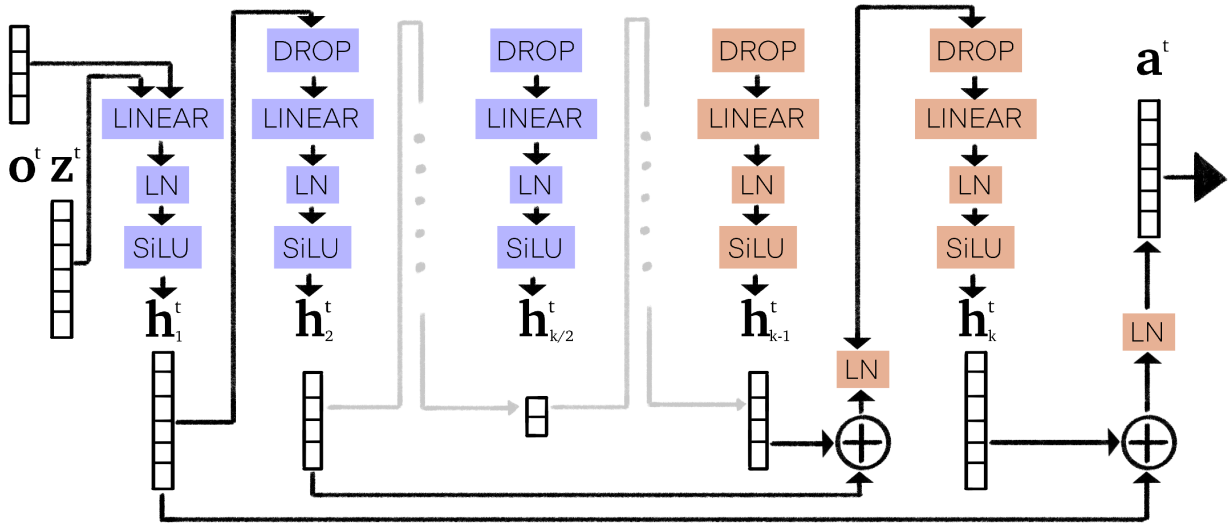


Figure 29 | Overview of UNET style 'synapse' recurrent models. $\mathbf{z}^t$ are the post-activations from the previous step, $\mathbf{o}^t$ are the attention outputs from observing data, and the synapse model yields $\mathbf{a}^t$ pre-activations for the NLMs to process. The UNET structure is set up so that the innermost bottleneck layer is 16 units wide, with linear scaling for each layer in between. Skip connections with layer-norm implement the classic UNET structure to maintain information. Layers that produce lower dimensionality are shown in blue, while layers that produce higher dimensionality are shown in orange.

B.2. Sampling synchronization neurons

The CTM operates using recurrence on a latent representation, $\mathbf{z}$, that is D-dimensional. $\mathbf{z}$ unfolds over time and the synchronization between *some chosen neurons* form the new kind of representation that the CTM enables.

There are $\frac{D\times(D+1)}{2}$ unique pairs of neurons, making for a substantially **larger set of neuron synchronization pairs** than there are neurons themselves. This motivates the need for a selection process. Over the development of the CTM we came up with three approaches to selecting neurons:

1. **Dense pairing**: in this setup we select J neurons and compute synchronization for every possible (i, j) pair of the J neurons. For $\mathbf{S}^t_{\text{out}}$ we choose J_{out} neurons and for $\mathbf{S}^t_{\text{action}}$ we choose non-overlapping J_{action} neurons. Selecting J_{out} neurons for dense pairing results in an output synchronization representation of $D_{\text{out}} = \frac{J_{\text{out}}\times(J_{\text{out}}+1)}{2}$, and similarly for the action representation. This approach essentially creates a strong bottleneck where all gradients must flow through the selected neurons, which can be advantageous for some tasks.
2. **Semi-dense pairing**: in this setup we open up the aforementioned bottleneck twofold by selecting two different subsets, J_1 and J_2, such that the left neurons of the synchronization dot product, i, are taken from J_1 and the right neurons, j, are taken from J_2. The same dense computation as before is then applied.
   The bottleneck width in this case is 2× as wide as before. Output and action selections are, once more, not overlapping.
3. **Random pairing**: in this setup we randomly select D_{out} or D_{action} pairs of neurons and compute the synchronization between each pair as opposed to doing so densely between all selected neurons. We also intentionally compute the (i, i) dot products between n_{self} neurons in each case in order to ensure that the CTM could recover a snapshot representation if it wanted to.

This opens up the bottleneck much more than before. We allow overlapping selections in this case.

C. ImageNet-1K

This section provides additional details and results for the ImageNet-1K experiments.

C.1. Architecture details

We used a constrained version of the classic ResNet architecture (He et al., 2016) for this task, which we adapted from https://github.com/huyvnphan/PyTorch_CIFAR10. It differs from the standard implementation for ImageNet in that the first convolution is constrained to use a kernel size of 3×3 as opposed to 7×7. We used a ResNet-152 structure and took the output prior to the final average pooling and projection to class logits. We used input images of size 224×224 which yielded 14×14 features for the keys and values used in cross attention.

We used the following hyperparameters:

- $D = 4096$ (the width of $\mathbf{z}^t$ and $\mathbf{a}^t$)
- $k = 16$ (synapse depth, 8 layers down and 8 layers up)
- $d_{\text{input}} = 1024$ (the width of attention output, $\mathbf{o}^t$)
- $n_{\text{heads}} = 16$
- Random pairing for neuron selection (see Appendix B.2)
- $D_{\text{out}} = 8196$ (width of $\mathbf{S}^t_{\text{out}}$ synchronization representation)
- $D_{\text{action}} = 2048$ (width of $\mathbf{S}^t_{\text{action}}$ synchronization representation)
- $n_{\text{self}} = 32$ (for recovering a snapshot representation)
- $T = 50$ (internal ticks)
- $M = 25$ (FIFO rolling memory input to NLMs)
- $d_{\text{hidden}} = 64$ (width of MLPs inside NLMs)
- $p_{\text{dropout}} = 0.2$ (dropout probability for synapse model)
- No positional embedding

We used the following settings for optimization:

- Trained using a batch size of 64 across 8 H100 Nvidia GPUs
- 500000 iterations for training, using a custom sampling such that each minibatch was sampled with possible replacement
- AdamW (Loshchilov and Hutter, 2017)
- A learning rate of 5e-4 with a linear warmup of 10000 iterations and decaying to zero using a cosine annealing learning rate scheduler
- Gradient norm clipping set to a norm of 20
- No weight decay

C.2. Loss function

Listing 5 shows the python code for the image classification loss function used to train the CTM on ImageNet-1K. This accompanies the loss defined in Section 2.5.

C.3. Additional demonstrations

D. 2D Mazes

D.1. Dataset

We used the maze-dataset repository to generate mazes for this work: https://github.com/understanding-search/maze-dataset. We generated mazes of size 19×19, 39×39, and 99×99.

```python
def image_classification_loss(predictions, certainties, targets, use_most_certain=True):
    """
    Computes the maze loss with auto-extending cirriculum.

    Predictions are of shape: (B, class, internal_ticks),
    Certainties are of shape: (B, 2, internal_ticks),
        where the inside dimension (2) is [normalised_entropy, 1-normalised_entropy]
    Targets are of shape: [B]

    use_most_certain will select either the most certain point or the final point.
    """
    targets_expanded = torch.repeat_interleave(targets.unsqueeze(-1), predictions.size(-1), -1)
    # Losses are of shape [B, internal_ticks]
    losses = nn.CrossEntropyLoss(reduction='none')(predictions, targets_expanded)

    loss_index_1 = losses.argmin(dim=1)
    loss_index_2 = certainties[:,1].argmax(-1)
    if not use_most_certain:  # Revert to final loss if set
        loss_index_2[:] = -1

    batch_indexer = torch.arange(predictions.size(0), device=predictions.device)
    loss_minimum_ce = losses[batch_indexer, loss_index_1].mean()
    loss_selected = losses[batch_indexer, loss_index_2].mean()

    loss = (loss_minimum_ce + loss_selected)/2
    return loss
```

Listing 5 | Loss function implementation of Section 2.5 for standard classification tasks, used for ImageNet-1K.

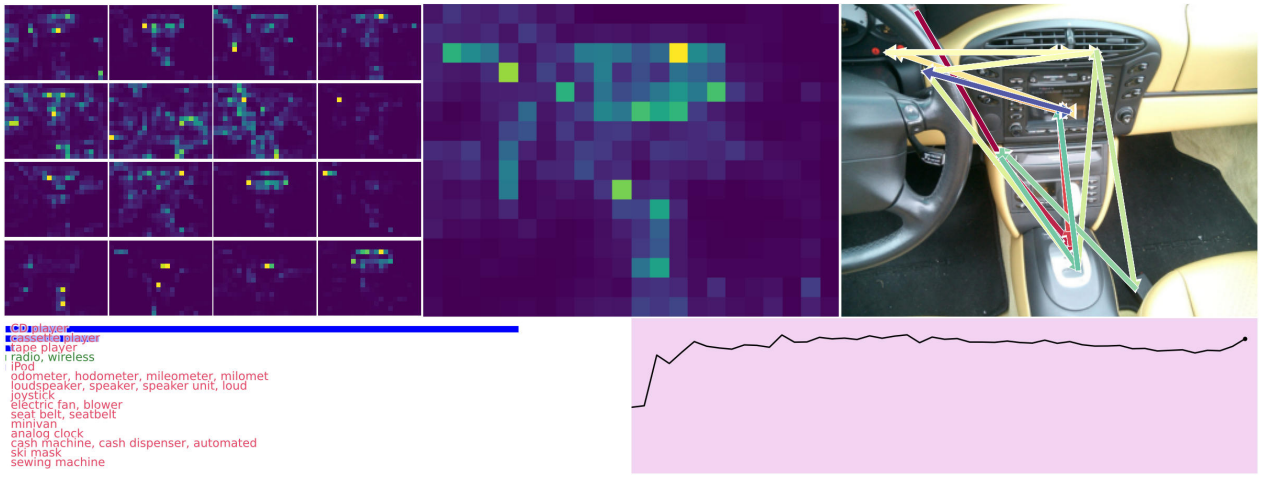


Figure 30 | Validation image index 1235, showing incorrect and uncertain prediction.

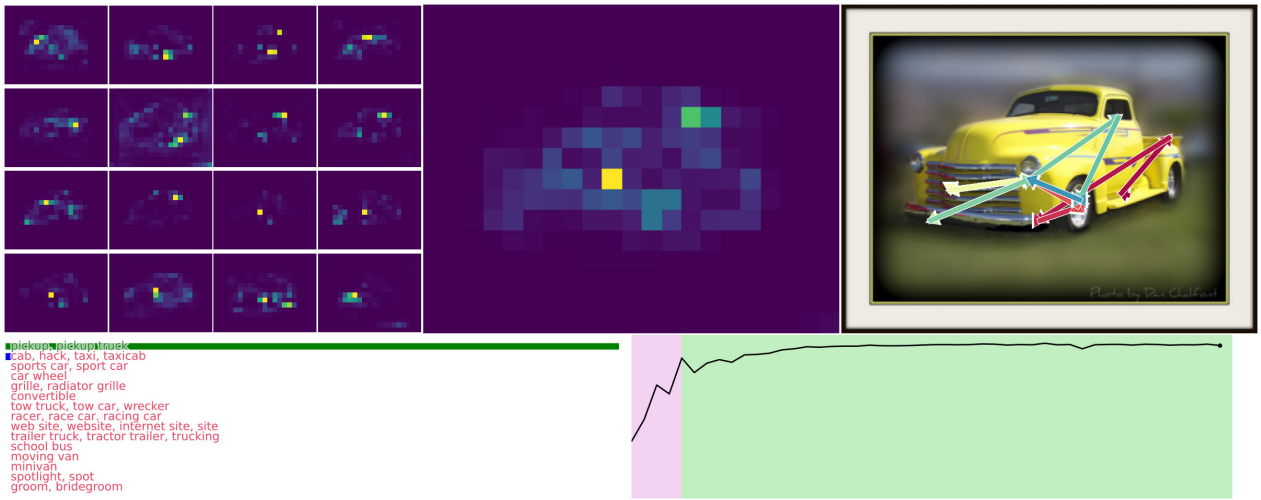


Figure 31 | Validation image index 15971, showing correct prediction and plausible 2nd most probable class.

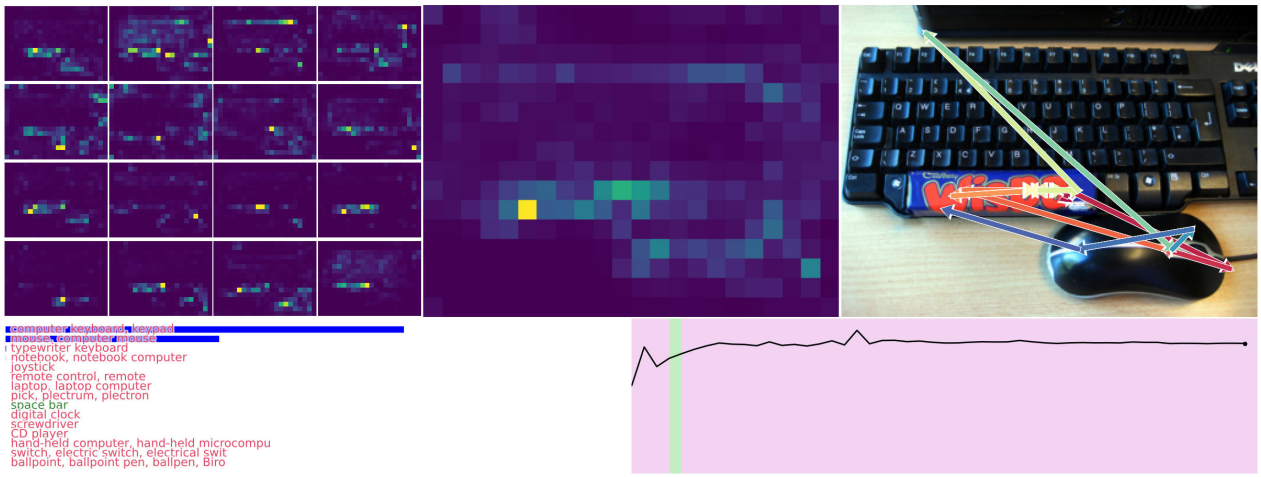


Figure 32 | Validation image index 21202, incorrect prediction after passing by correct prediction, showing ‘over-thinking’.

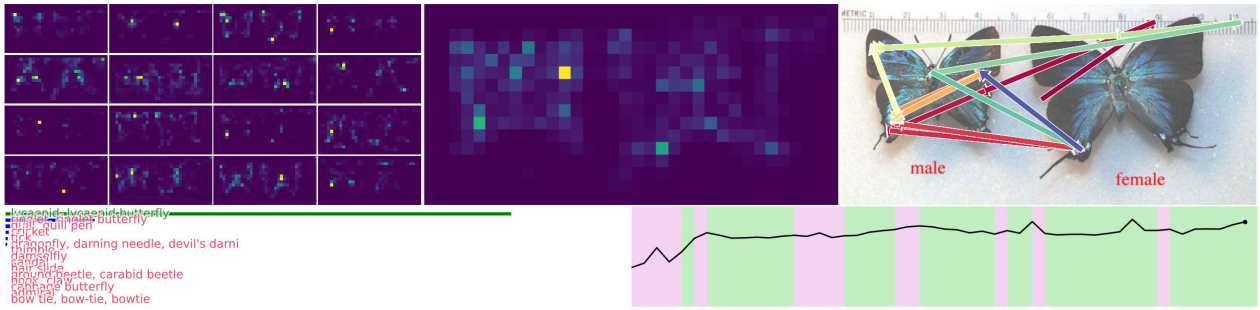


Figure 33 | Validation image index 39275, correct but uncertain prediction.

In each case we generated 50000 mazes and split them into train sets of size 45000 and test sets of size 5000. We used the 39×39 for training in this technical report and tested generalization on the 99×99. We provide all three maze datasets[8] in the CTM code repository.

D.2. Architecture details

We used the following hyperparameters:

- 39×39 mazes and a ResNet-34 backbone, where the keys and values were taken as features after the second hyper-block, resulting in a down-sample to 10×10
- $D = 2048$ (the width of $\mathbf{z}^t$ and $\mathbf{a}^t$)
- $k = 16$ (synapse depth, 8 layers down and 8 layers up)
- $d_{\text{input}} = 512$ (the width of attention output, $\mathbf{o}^t$)
- $n_{\text{heads}} = 16$
- Dense pairing for neuron selection (see Appendix B.2)
- $J_{\text{out}} = 32$ (width of $\mathbf{S}^t_{\text{out}}$ synchronization representation)
- $J_{\text{action}} = 32$ (width of $\mathbf{S}^t_{\text{action}}$ synchronization representation)
- $T = 75$ (internal ticks)
- $M = 25$ (FIFO rolling memory input to NLMs)
- $d_{\text{hidden}} = 32$ (width of MLPs inside NLMs)
- $p_{\text{dropout}} = 0.1$ (dropout probability for synapse model)
- No positional embedding

We used the following settings for optimization:

- Trained using a batch size of 64 on 1 H100 Nvidia GPU
- 1000000 iterations for training using AdamW (Loshchilov and Hutter, 2017)
- A learning rate of 1e-4 with a linear warmup of 10000 iterations and decaying to zero using a cosine annealing learning rate scheduler
- No weight decay

This resulted in a model with 31,998,330 parameters.

D.3. Maze curriculum

We adapted the loss function for solving mazes to include a curriculum element. Before computing t_1 and t_2 for the loss in Equation (12) we first altered the **loss at each internal tick** to only account for those steps in the maze that were correctly predicted with plus 5 additional steps along the path. This effectively lets the model (CTM or LSTM baselines) slowly learn to solve the maze from start to finish. Listing 6 shows how this is implemented when computing the loss and is used for both CTM and LSTM training.

D.4. Baselines details

We tested a number of LSTM baselines for solving this task, but struggled with stability during training (see Figure 7), particularly beyond a single LSTM layer or with a higher number of internal ticks. Hence we tested three LSTM configurations of depths 1,2, and 3. For each model we tested with 75 internal ticks to match the CTM, and 50 internal ticks for stability. We also tested a feed-forward model, projecting the feature space (before average pooling) into a hidden layer of the same width as the CTM, yielding a slightly higher parameter count. We kept all hyperparameters constant, yielding the following setups:

- **LSTM**, 1 layer, $T = 50$ and $T = 75$: 42,298,688

[8] We found the 19×19 beneficial for debugging, hence we provide it too.

```python
def image_classification_loss(predictions, certainties, targets, use_most_certain=True):
    """
    Computes the maze loss with auto-extending cirriculum.

    Predictions are of shape: (B, class, internal_ticks),
    Certainties are of shape: (B, 2, internal_ticks),
        where the inside dimension (2) is [normalised_entropy, 1-normalised_entropy]
    Targets are of shape: [B]

    use_most_certain will select either the most certain point or the final point.
    """
    targets_expanded = torch.repeat_interleave(targets.unsqueeze(-1), predictions.size(-1), -1)
    # Losses are of shape [B, internal_ticks]
    losses = nn.CrossEntropyLoss(reduction='none')(predictions, targets_expanded)

    loss_index_1 = losses.argmin(dim=1)
    loss_index_2 = certainties[:,1].argmax(-1)
    if not use_most_certain:  # Revert to final loss if set
        loss_index_2[:] = -1

    batch_indexer = torch.arange(predictions.size(0), device=predictions.device)
    loss_minimum_ce = losses[batch_indexer, loss_index_1].mean()
    loss_selected = losses[batch_indexer, loss_index_2].mean()

    loss = (loss_minimum_ce + loss_selected)/2
    return loss
```

Listing 6 | Loss function implementation of Section 2.5 for maze route prediction, including an auto curriculum approach for both CTM and LSTM.

- **LSTM**, 2 layers, $T = 50$ and $T = 75$: 75,869,504 parameters
- **LSTM**, 3 layers, $T = 50$ and $T = 75$: 109,440,320 parameters
- **Feed-forward**, with a hidden layer width of 2048 (and GLU activation thereon): 54,797,632 parameters

D.5. Maze loss curves

Figure 34 gives the loss curves for the maze solving models in Section 4.1, showing how the CTM is more stable and performant when trained on this task.

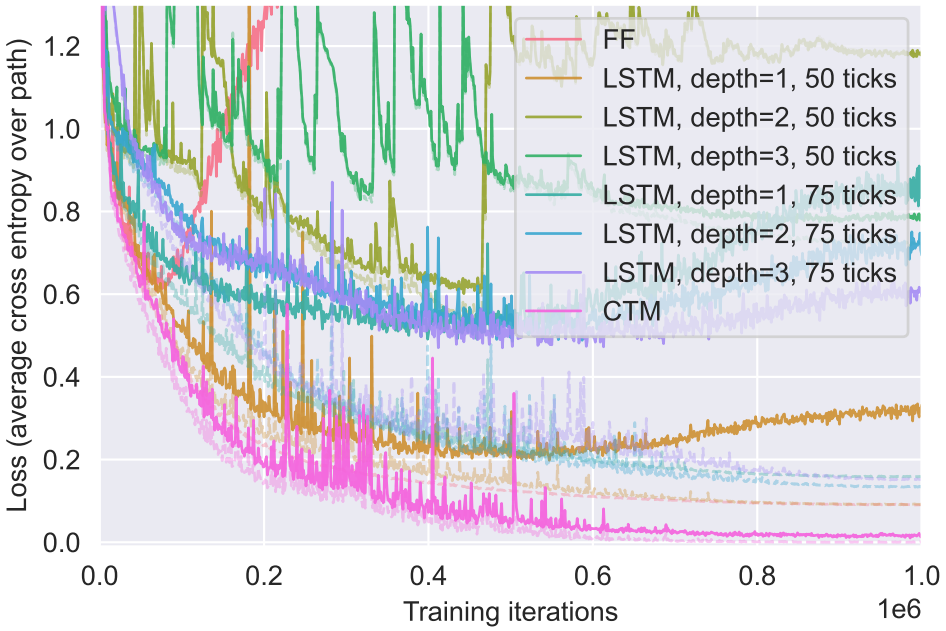


Figure 34 | Loss curves when training the CTM and baselines.

E. CIFAR-10 versus humans

We used a the first hyper-block of a constrained ResNet-18 backbone (see Appendix C.1): 5 convolutional layers in total and a downsample factor of 2×. We used the following hyperparameters for the CTM:

- $D = 256$ (the width of $\mathbf{z}^t$ and $\mathbf{a}^t$)
- $k = 10$ (synapse depth, 5 layers down and 5 layers up)
- $d_{\text{input}} = 64$ (the width of attention output, $\mathbf{o}^t$)
- $n_{\text{heads}} = 16$
- Random pairing for neuron selection (see Appendix B.2)
- $D_{\text{out}} = 256$ (width of $\mathbf{S}^t_{\text{out}}$ synchronization representation)
- $D_{\text{action}} = 512$ (width of $\mathbf{S}^t_{\text{action}}$ synchronization representation)
- $n_{\text{self}} = 0$
- $T = 50$
- $M = 15$ (FIFO rolling memory input to NLMs)
- $d_{\text{hidden}} = 64$ (width of MLPs inside NLMs)
- $p_{\text{dropout}} = 0.0$
- Weight decay of 0.0001
- No positional embedding

We used the following settings for optimization:

- Trained using a batch size of 512 on 1 H100 Nvidia GPU
- 600000 iterations for training using AdamW (Loshchilov and Hutter, 2017)
- A learning rate of 1e-4 with a linear warmup of 2000 iterations and decaying to zero using a cosine annealing learning rate scheduler

For the LSTM baseline a 2-layer LSTM was used as this performed better than a single layer LSTM setup and was relatively stable in training (compared to the maze task). The CTM synapse depth, memory length, and NLM hidden width were chosen such that the model width was kept constant (256) while parameter counts were closely matched between the CTM and LSTM. For the feed-forward model we kept the model width constant.

F. CIFAR-100

This section discusses the details of the CIFAR-100 experiments.

F.1. Architecture details

For Section 6.1 we used a the first two hyper-blocks of a constrained ResNet-34 backbone (see Appendix C.1): a downsample factor of 4×. For this experiment we varied D but kept all other hyperparameters as:

- $k = 8$ (synapse depth, 4 layers down and 4 layers up)
- $d_{\text{input}} = 512$ (the width of attention output, $\mathbf{o}^t$)
- $n_{\text{heads}} = 8$
- Random pairing for neuron selection (see Appendix B.2)
- $D_{\text{out}} = 2048$ (width of $\mathbf{S}^t_{\text{out}}$ synchronization representation)
- $D_{\text{action}} = 1024$ (width of $\mathbf{S}^t_{\text{action}}$ synchronization representation)
- $n_{\text{self}} = 32$
- $T = 50$
- $M = 25$ (FIFO rolling memory input to NLMs)
- $d_{\text{hidden}} = 32$ (width of MLPs inside NLMs)

- $p_{\text{dropout}} = 0.2$
- No weight decay
- No positional embedding

For Section 6.2 we used the first two hyper-blocks of a constrained ResNet-19 backbone (see Appendix C.1): a downsample factor of 4×. For this experiment we varied T but kept all other hyperparameters as:

- $D = 512$
- $k = 4$ (synapse depth, 2 layers down and 2 layers up)
- $d_{\text{input}} = 256$ (the width of attention output, $\mathbf{o}^t$)
- $n_{\text{heads}} = 4$
- Random pairing for neuron selection (see Appendix B.2)
- $D_{\text{out}} = 256$ (width of $\mathbf{S}^t_{\text{out}}$ synchronization representation)
- $D_{\text{action}} = 256$ (width of $\mathbf{S}^t_{\text{action}}$ synchronization representation)
- $n_{\text{self}} = 0$
- $M = 25$ (FIFO rolling memory input to NLMs)
- $d_{\text{hidden}} = 16$ (width of MLPs inside NLMs)
- $p_{\text{dropout}} = 0.0$
- Weight decay of 0.001
- No positional embedding

This model was set up to be more constrained compared to the other CIFAR-100 ablation because of the overhead induced by using more ticks. We explained in Section 6.2 that the variants trained with longer ticks could benefit from more training, and this disparity would be greater should we use bigger models for this experiment.

G. Parity

G.1. Dataset details

The input data for the parity task is a vector of length 64, where at each position is either a −1 or 1. The target for each sample is a vector of the same size, where at each position is the parity of the sequence up to that position, which we refer to as the cumulative parity. This data is generated on the fly, each time a new batch is fetched.

G.2. Architecture details

The following architecture is used for the experiments in the parity task. Inputs to the model are of shape $(B, 64)$, where the size of the minibatch and sequence length are B and 64, respectively. Each of the values in the 64-length sequence are either -1 or 1, at random. First, the values of -1 and 1 are converted into embeddings in $d_{\text{embed}} = d_{\text{input}}$ and positional embeddings are added. The resulting embeddings are passed through a linear layer with layer normalization to form (identical) attention keys and values. As described in section 2, the synchronization between J_{action} neurons is computed and from this representation an attention query is formed. This query is then used to compute the attention values, which are concatenated to the activated state to be processed by the synapses and the neuron-level models. For the synapses we use a shallow feedforward network. This process repeats for T internal ticks, where at each internal tick t, the synchronization can be computed between J_{out} neurons and projected to the logit space.

In the parity task, we experimented with how the model performs with a varying number of internal ticks and memory length. As a baseline, we use single-layer LSTMs which are both parameter matched and use the same number of internal ticks as the CTM.

All CTM models share a common set of architectural hyperparameters, listed below. The Table 3 shows the subset of hyperparameters that vary across experimental configurations.

- $d_{\text{model}} = 1024$
- $d_{\text{input}} = 512$
- $d_{\text{hidden}} = 4$
- $k = 1$
- $p_{\text{dropout}} = 0$
- $n_{\text{heads}} = 8$
- $J_{\text{action}} = 32$
- $J_{\text{out}} = 32$
- Semi-dense pairing was used for selecting neurons for synchronization
- Absolute positional encoding was added to the input features

Model	T	M	d_{model}	**Total Parameters**
CTM	1	1	1024	4908706
LSTM	1	–	669	4912710
CTM	10	5	1024	5043874
LSTM	10	–	686	5050716
CTM	25	10	1024	5212834
LSTM	25	–	706	5224386
CTM	50	25	1024	5719714
LSTM	50	–	765	5722374
CTM	75	25	1024	5719714
LSTM	75	–	765	5722374
CTM	100	50	1024	6564514
LSTM	100	–	857	6567486

Table 3 | Model hyperparameters for the parity task (that vary across configurations).

G.3. Optimization details

The CTM was trained using the certainty-based loss function described in section 2.5, whereas the LSTM baselines utilized the cross-entropy loss computed at the final internal tick. This choice was made due to initial difficulties in training the LSTM effectively with the certainty-based loss. In Figure 35, we compare training accuracy curves for the LSTM baselines with 10 and 25 iterations, trained with either the final or certainty-based loss. Generally, both loss functions lead to unstable training for LSTMs with multiple internal ticks.

We used the following settings for optimization:

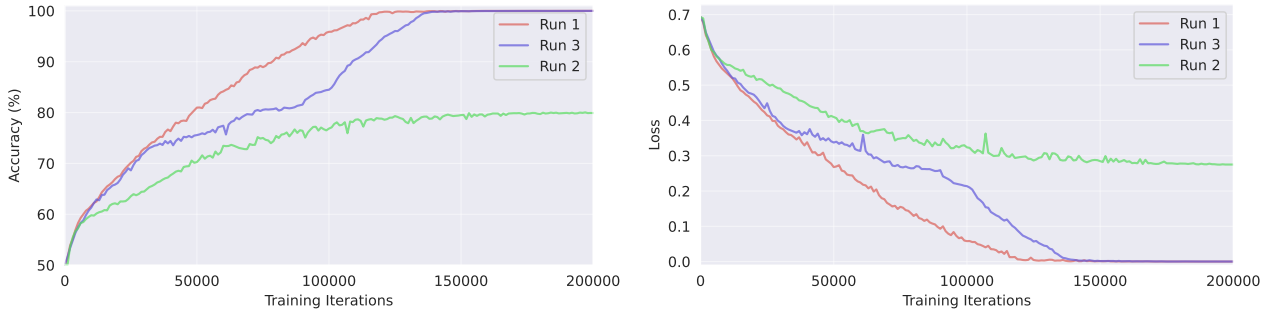
- Trained using a batch size of 64 on 1 H100 Nvidia GPU.
- 200000 iterations for training using AdamW (Loshchilov and Hutter, 2017).
- A learning rate of 1e-4 with a linear warmup of 500 iterations and decaying to zero using a cosine annealing learning rate scheduler.

G.4. Results

Model performance varies significantly between seeds Figure 19 shows the accuracy over training for various CTM and LSTM configurations, with each configuration averaged over three independent runs. These training curves exhibit considerable variance due to significant differences in performance between runs, strongly influenced by the initial random seed. For example, Figure 36 shows individual

Figure 35 | Test accuracies for LSTM baselines, trained with either the certainty-based loss or the cross-entropy loss at the final internal tick. Both loss functions lead to unstable learning.

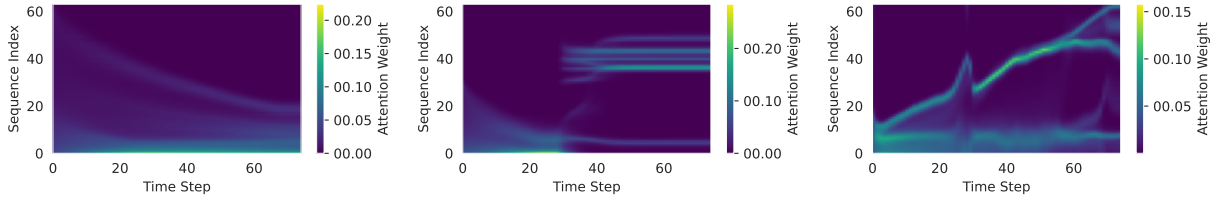
training curves for the CTM trained with 75 internal ticks and a memory length of 25. Runs 1 and 3 reach perfect accuracy, while run 2 converges to a suboptimal solution.



(a) Accuracy. **(b)** Loss.

Figure 36 | Training curves for three CTMs trained with three random seeds. Run 1 and 3 converge to a loss of zero, while the other run converges to a non-zero loss.

Furthermore, all three of these models display significantly different behaviors, with each CTM attending to very different parts of the input sequence over the 75 internal ticks. These attention patterns over the internal ticks are shown in Figure 37. Run 3 results in a model that attends from the beginning to the end of the entire sequence, while run 1 attends in reverse order.



(a) Run 1. **(b)** Run 2. **(c)** Run 3.

Figure 37 | Attention patterns for each of the three runs after training.

H. Q&A MNIST

H.1. Architecture details

Unlike other tasks, the Q&A MNIST task processes multiple input types: MNIST digit images, embeddings for operator and index markers, and zero tensors as answer flags. MNIST images undergo

preprocessing through a convolutional backbone consisting of two convolutional blocks, each containing a convolutional layer, batch normalization, a ReLU activation, and a max pooling layer. Outputs from this backbone form attention keys and values, which the CTM queries using projections from the synchronization representation. The resulting attention outputs are concatenated with the CTM's activated state before synaptic processing. In contrast, operator and index embeddings, as well as answer flags, bypass the convolutional backbone and attention mechanism, being directly concatenated to the CTM's activated state. Operators use learned embeddings, indices utilize sinusoidal embeddings (Vaswani et al., 2017), and answer flags are zero vectors matching the embedding dimension.

For comparison, parameter and internal tick matched single-layer LSTM baselines were used. The common parameters used in the experiment are as follows:

- $d_{\text{model}} = 1024$
- $d_{\text{input}} = 64$
- $d_{\text{hidden}} = 16$
- $k = 1$
- $p_{\text{dropout}} = 0$
- $n_{\text{heads}} = 4$
- $J_{\text{action}} = 32$
- $J_{\text{out}} = 32$
- Semi-dense pairing was used for selecting neurons for synchronization.
- Positional encoding was not used.

Detailed hyperparameters of the CTM are provided in Table 4.

Model	T	M	Repeats/Input	Answering Steps	Total Parameters
CTM	1	3	1	1	2,501,388
LSTM	1	–	1	1	2,507,218
CTM	10	30	10	10	3,413,772
LSTM	10	–	10	10	3,418,954

Table 4 | Differing model hyperparameters and total parameters for the Q&A MNIST experiments. The column **Repeats/Input** refers to the number of internal ticks the model used to process a unique input. For example, **Repeats/Input** = 10 implies that 10 internal ticks are used to process each MNIST digit and each index or operator embedding. The **Answering Steps** refers to the number of internal ticks the answering flag is observed for.

H.2. Optimization details

The CTM was trained using the certainty-based loss function described in section 2.5, while the LSTM baselines were trained using the cross-entropy loss at the final internal tick. We used the following settings for optimization:

- Trained using a batch size 64 on 1 H100 Nvidia GPU.
- 300000 iterations for training using AdamW (Loshchilov and Hutter, 2017).
- A learning rate of 1e-4 with a linear warmup of 500 iterations and decaying to zero using a cosine annealing learning rate scheduler.

I. Reinforcement learning

I.1. Environment details

CartPole. The CartPole task (*CartPole-v1*) is a classic task in reinforcement learning. It involves balancing a pole, which is hinged to a cart moving along a frictionless track. The system is controlled

by applying horizontal forces (left or right) to the cart, with the objective of keeping the pole upright. A reward of +1 is given for every step taken, with the episode terminating when either the pole angle is greater than $\pm 12^\circ$, the cart position is greater than ± 2.4, or the episode length is greater than the maximum number of steps, which we set to 200 steps. Furthermore, we use reward normalization, such that the exponential moving average of immediate rewards has an approximately fixed variance.

The action space is composed of 2 discrete actions, corresponding to the direction of force applied to the cart. In the typical cartpole task, the observation space is of shape $(4,)$ with values corresponding to the cart position, cart velocity, pole angle and pole angular velocity. For our experiments with the CTM, however, the cart velocity and the pole angular velocity are masked to make the environment partially observable.

Acrobot. In the Acrobot task (*Acrobot-v1*), a system composed of two links connected linearly to form a chain, with one end of the chain fixed. The joint between the two links is actuated, while the joint at the fixed end is free to rotate. The goal is to apply torques to the actuated joint to swing the free end of the chain above a certain height in as few steps as possible, with the chain initially hanging down with some initial random angle and velocity. An episode ends when the chain reaches above the required height, or a maximum number of steps are taken, which we set to 500 steps. A reward of -1 is incurred for all steps taken to reach the goal, with a limit of -100.

The action space consists of three discrete actions, which are to apply -1, 0 and 1 Nm of torque to the actuated joint. The observation space is of shape (6,) composed of the sine and cosine of the angle made by the first joint (such that an angle of 0 indicated the first link is pointing directly downwards), the sine and cosine of the second link (such that an angle of 0 corresponds to having the same angle between the two links), and the angular velocity of two angles. As in the CartPole task, these two angular velocity components are masked such that the environment is only partially observable.

MiniGrid Four Rooms. The MiniGrid Four Rooms task (*MiniGrid-FourRooms-v0*) is a reinforcement learning environment in which an agent must navigate in a grid world composed of four rooms interconnected by four gaps in the walls. The agent receives a reward of $1 - 0.9(\text{step count} \times \text{max steps})$ if it reaches the goal located at the green square, or 0 otherwise. Again, reward normalization is used to normalize the moving average of past rewards. The agent's position, the goal position, and each of the four gaps in the walls are positioned randomly at the start of each episode. The environment terminates when the agent reaches the goal or when the maximum number of steps has been reached, which we set to 300 steps.

The action space is composed of 7 discrete actions, corresponding to turn left, turn right, go forward, pickup, drop, toggle, and done. Of these actions, only the first three are relevant to the task, with the other four actions behaving as a wait or no-op action. In this task, the agent has a limited field of view, consisting of the 7×7 grid located in front of the agent. The observation space of this environment is of shape $7 \times 7 \times 3$, where each of the 7×7 tiles is encoded as a three-dimensional tuple corresponding to the object, color and state IDs at that position. Specifically, there are 11 different objects (such as wall, floor, etc.), 6 different colors and 3 different states (open, closed, etc.).

I.2. Architecture details

The configuration of the CTM for training with PPO is as follows. First, observations are processed by a backbone, such as a feedforward network, and are concatenated to the current activated state of the CTM, without an attention mechanism, for processing over a fixed number of internal ticks. After this fixed number of internal ticks, the synchronization between the output neurons is calculated and passed to the actor and critic heads for selecting the next action and estimating the state value.

Unlike the other tasks, which calculate synchronization across the entire activation history, in the RL setting we use a sliding window of size memory length M. This approach prevents the buildup of very long activation histories, which may grow into the thousands for these tasks. Additionally, this allows for maintaining tensors of the same shape across all stages of the episode rollouts. To facilitate this, the CTM is initialized with both a learned initial state trace and a learned initial activated state trace, which are supplied to the model on the initialization of each episode. After a single forward pass of the model (corresponding to a single environment step), these state traces will be maintained and provided to the model on the next environment step. This allows the CTM to process a continuous history of activity, enabling activations from many environment states in the past to impact the present.

For the classic control tasks, the observation backbone is composed of two blocks containing a linear layer, a gated linear unit (GLU) (Dauphin et al., 2017) and layer normalization. A similar input processing is carried out for the navigation task, however, each of the object, color and state IDs are first embed in $d_{embed} = 8$. While for the CTM the output of the backbone is concatenated to the current activated state, for the LSTM baseline, the output is instead processed for the same number of internal ticks as the CTM. The actor and critic heads are implemented as two-layer multilayer perceptrons (MLPs), each comprising two hidden layers of 64 neurons with ReLU activations. For the CTM, these heads receive the synchronization of the output neurons as inputs, while for the LSTM baselines, they receive the hidden state of the LSTM after T internal tick. Dense pairing was used to select neurons for synchronization.

Unlike other tasks such as image classification (Section 3), which use a UNet-style synapse model, the RL tasks employ a two-layer feedforward synapse, where each layer consists of a linear transformation, a GLU and LayerNorm. Empirically, we found that two of these layers significantly outperformed a single layer, particularly in the navigation task, where a single-layer synapse consistently failed to match the LSTM's average episode length.

The model hyperparameters used for the experiments for CartPole, Acrobot and MiniGrid Four Rooms can be found in Tables 5 to 7.

Model	T	M	d_{model}	d_{input}	d_{hidden}	J_{out}	**Total Parameters**
CTM	1	10	128	128	4	16	175437
LSTM	1	–	118	128	–	–	175855
CTM	2	20	128	128	4	16	188237
LSTM	2	–	126	128	–	–	188863
CTM	5	50	128	128	4	16	226637
LSTM	5	–	148	128	–	–	227275

Table 5 | Model hyperparameters for the CartPole experiments.

Model	T	M	d_{model}	d_{input}	d_{hidden}	J_{out}	**Total Parameters**
CTM	1	5	256	64	4	16	350094
LSTM	1	–	243	64	–	–	350118
CTM	2	10	256	64	4	16	362894
LSTM	2	–	249	64	–	–	364290
CTM	5	25	256	64	4	16	401294
LSTM	5	–	265	64	–	–	403490

Table 6 | Model hyperparameters for the Acrobot experiments.

Model	T	M	d_{model}	d_{input}	d_{hidden}	J_{out}	Total Parameters
CTM	1	10	512	128	16	32	7802690
LSTM	1	–	294	128	–	–	7813692
CTM	2	20	512	128	16	32	7976770
LSTM	2	–	300	128	–	–	7979304

Table 7 | Model hyperparameters for the MiniGrid Four Rooms experiments.

I.3. Optimization details

The models were trained with Proximal Policy Optimization (Schulman et al., 2017) on single H100 Nvidia GPU. The same set of PPO hyperparameters were used for both the CTM and the LSTM baseline, and are shown in Table 8.

Hyperparameter	CartPole	Acrobot	MiniGrid Four Rooms
Learning Rate (LR)	1×10^{-3}	5×10^{-4}	1×10^{-4}
Total Environment Steps	10M	2M	300M
Rollout Length	50	100	50
Number of Environments	256	12	256
Max Environment Steps per Episode	200	500	300
Update Epochs	4	1	1
Minibatches	4	4	4
Discount Factor (γ)	0.99	0.99	0.99
GAE Lambda (λ)	0.95	0.95	0.95
Clip Coefficient	0.1	0.1	0.1
Entropy Coefficient	0.1	0.1	0.1
Value Function Coefficient	0.25	0.25	0.25
Value Function Clipping	No	No	No
Max Gradient Norm	0.5	0.5	0.5

Table 8 | PPO hyperparameters for each task.

J. UMAP

We used UMAP (McInnes et al., 2018) to build Figure 6. The purpose of UMAP in this case was to give each neuron in the ImageNet CTM a 2D location such that when their activities over time were visualized a meaningful pattern could be observed, should it exist. To this end, we considered the histories of post-activations for 200 different images as the high-dimensional inputs to UMAP ($200 \times T = 200 \times 5 = 1000$ dimensions). UMAP was then used to project this to a 2D space for visualization.

K. Recursive computation of the synchronization matrix

In Section 2.4 we defined the synchronization matrix at internal tick t as

$$\mathbf{S}^t = \mathbf{Z}^t (\mathbf{Z}^t)^\intercal, \qquad \mathbf{Z}^t \in \mathbb{R}^{D \times t}, \tag{13}$$

where the d–th row of $\mathbf{Z}^t$ stores the post–activation trace of neuron d up to tick t (cf. Eq. (4)). Because Eq. (13) recomputes all D^2 inner products from scratch at every tick, its time complexity is $O(D^2 t)$ over a roll-out of length t. Below we show that, with the exponentially–decaying rescaling of Eq. (10), the same quantity can be obtained from a pair of first–order recursions that require only $O(D_{\text{sub}})$ work per tick, where $D_{\text{sub}} \ll D$ is the number of subsampled neuron indices actually used for the output and action projections.

For notational clarity we first consider a single (i, j) neuron pair and omit the subsampling; the extension to a batch of pairs is immediate. Recall that the rescaled synchronization entry is defined as

$$S_{ij}^t = \frac{\sum_{\tau=1}^{t} \mathrm{e}^{-r_{ij}(t-\tau)}\, z_i^\tau\, z_j^\tau}{\sqrt{\sum_{\tau=1}^{t} \mathrm{e}^{-r_{ij}(t-\tau)}}}, \tag{14}$$

where $r_{ij} \geq 0$ is the learnable decay rate for the pair (i, j). Define the following auxiliary sequences

$$\alpha_{ij}^t := \sum_{\tau=1}^{t} \mathrm{e}^{-r_{ij}(t-\tau)}\, z_i^\tau\, z_j^\tau, \qquad \alpha_{ij}^1 = z_i^1 z_j^1, \tag{15}$$

$$\beta_{ij}^t := \sum_{\tau=1}^{t} \mathrm{e}^{-r_{ij}(t-\tau)}, \qquad \beta_{ij}^1 = 1. \tag{16}$$

Then $S_{ij}^t = \alpha_{ij}^t / \sqrt{\beta_{ij}^t}$ and both α_{ij}^t and β_{ij}^t obey simple first–order difference equations:

$$\alpha_{ij}^{t+1} = \mathrm{e}^{-r_{ij}}\, \alpha_{ij}^t \;+\; z_i^{t+1}\, z_j^{t+1}, \tag{17}$$

$$\beta_{ij}^{t+1} = \mathrm{e}^{-r_{ij}}\, \beta_{ij}^t + 1. \tag{18}$$

The rank–1 update in Eq. (17) makes it unnecessary to store the full activation history or to repeatedly form large outer products. During forward simulation we maintain α_{ij}^t and β_{ij}^t for each selected pair and update them in $O(1)$ time.

In practice we store $\{\alpha_{ij}^t, \beta_{ij}^t\}$ only for the two disjoint subsamples that form $\mathbf{S}_{\text{out}}^t$ and $\mathbf{S}_{\text{action}}^t$ (Section 2.4). Both memory footprint and compute overhead therefore scale linearly with the number of retained pairs, i.e. $O(D_{\text{sub}}) = O(D_{\text{out}} + D_{\text{action}})$ per tick.